4

Hierarchical Process Composition

Dynamic Maintenance of Structure in a Distributed Environment

Stuart Arthur Friedberg

Technical Report 294
1988

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Hierarchical Process Composition
Dynamic Maintenance of Structure in a Distributed Environment

by

Stuart Arthur Friedberg

Submitted in Partial Fulfillment
of the
Requirements for the Degree

Doctor of Philosophy

Supervised by Thomas J. LeBlanc

Department of Computer Science
College of Arts and Sciences

University of Rochester

Rochester, New York

1988

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 294	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Hierarchical Process Composition: Dynamic Maintenance of Structure in a Distributed Environment		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Stuart Arthur Friedberg		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0197 N00014-82-K-0193 DACA76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department 734 Computer Studies Bldg. University of Rochester, Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS D. Adv. Res. Proj. Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE 1988 13. NUMBER OF PAGES 163
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Res. Information Systems Arlington, VA 22217 US Army, ETL Fort Belvoir, VA 22060		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution of this document is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES None.		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Reconfiguration, networks, interprocess communication, access control, consistency		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This dissertation is a study in depth of a method, called Hierarchical Process Composition (HPC), for organizing, developing, and maintaining large distributed programs. HPC extends the process abstraction to nested collections of processes, allowing a multiprocess program in place of any single process, and provides a rich set of structuring mechanisms for building distributed/software systems, especially interactions involving dynamic reconfiguration, protection, and distribution. The major contributions of this work come from the detailed consideration, based on case studies, formal analysis,		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(insert): applications. The emphasis in HPC is on structural and architectural issues in distributed

20. ABSTRACT (Continued)

and a prototype implementation, of how abstraction and composition interact in unexpected ways with each other and with a distributed environment.

HPC ties processes together with heterogenous interprocess communication mechanisms, such as TCP/IP and remote procedure call. Explicit structure determines the logical connectivity between processes, masking differences in communication mechanisms. HPC supports one-to-one, parallel channel, and many-to-many (multicasting) connectivity. Efficient computation of end-to-end connectivity from the communication structure is a challenging problem, and a third-party connection facility is needed to implement dynamic reconfiguration when the logical connectivity changes.

Explicit structure also supports grouping and nesting of processes. HPC uses this process structure to define meaningful protection domains. Access control is structured (and the basic HPC facilities may be extended) using the same powerful tools used to define communication patterns. HPC provides escapes from the strict hierarchy for direct communication between any two programs, enabling transparent access to global services. These escapes are carefully controlled to prevent interference and to preserve the appearance of a strict hierarchy.

This work is also a rare case study in consistency control for non-trivial, highly-available services in a distributed environment. Since HPC abstraction and composition operations must be abailable during network partitions, basic structural constraints can be violated when separate partitions are merged. By exhaustive case analysis, all possible merge inconsistencies that could arise in HPC have been identified and it is shown how each inconsistency can be either avoided, automatically reconciled by the system, or reported to the user for application-specific reconciliation.

Curriculum Vitae

Stuart Arthur Friedberg was born in Bloomington, Indiana, on 7 November 1958. Following high school at Howe Military School, he joined the Lloyd House at the California Institute of Technology, intending to major in biochemistry. Culture shock cut his tenure at Caltech short, but not before he discovered the joys of computing. Fortunately, his friends weaned him from a diet of Fortran and Macro-10 onto Pascal before he returned home from California.

Lacking academic credentials and work experience, Friedberg paid a visit to a traditional employer of last resort, enlisting in the United States Air Force as a Voice Processing Specialist (AFSC 20873C) in August 1977. Following a year's training in Czech and Slovak at the Defense Language Institute - Foreign Language Center in Monterey, California, he joined the 6913th Electronic Security Squadron in Augsburg, Federal Republic of Germany. He was later selected for an assignment to the 6941st ESS at the National Security Agency, Fort George G. Meade, Maryland. Before mustering out of the service in August 1982, Friedberg was awarded the Meritorious Service Medal and the Joint Service Commendation Medal for outstanding performance of duties.

During his military service, Friedberg was preparing for a permanent career in computer science. Forced to substitute books for unprincipled bit-twiddling, and enjoying the flexible schedule of an independent student, he obtained the degree Bachelor of Science from the Regents of New York External Degree Program in May 1981.

Friedberg joined the Computer Science Department at the University of Rochester immediately after leaving the Air Force. Under the initial aegis of Jerome Feldman, he earned the degree Master of Science in May 1983, passing the qualifying exams in the previous year. In this first two years at Rochester, he worked as both teaching and research assistant, and as an independent consultant to the Xerox Corporation in Webster, New York.

After examining earlier Activities work, Friedberg joined with his thesis advisor, Thomas LeBlanc, to investigate structuring systems for multiprocess, distributed programs. A thesis paper presented at the 5th International Conference on Distributed Computing Systems in May 1985 was awarded "Best Paper" for that conference.

Friedberg will join the faculty of the Computer Science Department at the University of Wisconsin - Madison in August 1988, with award of the degree Doctor of Philosophy from the University of Rochester in October 1988.

Acknowledgements

It has become cliche to acknowledge the positive environment provided by a first-class research department, but I have greater reason than most to make such acknowledgement. Rochester accepted me as a graduate student despite a non-traditional background (disasterous college record, no letters of recommendation from academics, no industrial experience, publications, no track record of research) that triggered immediate rejections at many other institutions. This prompts a certain appreciation.

In the last six years, the department has provided every reasonable resource, and a few unreasonable ones, I needed to concentrate on my work. The administrative staff deserve many kudos for keeping things running smoothly, despite the faculty and students. My fellow graduate students have been an invaluable learning resource (especially for getting through theory in the first year). I thank them all, because the list of thoughtful conversationalists must include half the department.

The initial IPC router process was implemented by Rochester senior Derek Pitcher (now with Lockheed) and much of the client-kernel interface was implemented by fellow graduate student Brian Marsh. John Costanzo wrote a special data management package for this thesis.

On the faculty, Chris Brown did his usual painstaking job of reading horrible early drafts and suggesting how they might be converted into English. My advisor, Tom LeBlanc, deserves the phrase long-suffering, contending with both stylistic and technical matters in two completely different thesis documents, and uncounted iterations on most things. I would also like to take this opportunity to publically apologize for shouting so much in his office.

This work has been supported in part by ONR research contract number N00014-80-C-0197, in part by U.S. Army Engineering Topographic Laboratories research contract number DACA76-85-C-0001, in part by NSF Coordinated Experimental Research grant number DCR-8320136, and in part by ONR/DARPA research contract number N00014-82-K-0193. I thank the Xerox Corporation University Grants Program for providing equipment used in the preparation of this paper.

Abstract

This dissertation is a study in depth of a method, called Hierarchical Process Composition (HPC), for organizing, developing, and maintaining large distributed programs. HPC extends the process abstraction to nested collections of processes, allowing a multiprocess program in place of any single process, and provides a rich set of structuring mechanisms for building distributed applications. The emphasis in HPC is on structural and architectural issues in distributed software systems, especially interactions involving dynamic reconfiguration, protection, and distribution. The major contributions of this work come from the detailed consideration, based on case studies, formal analysis, and a prototype implementation, of how abstraction and composition interact in unexpected ways with each other and with a distributed environment.

HPC ties processes together with heterogenous interprocess communication mechanisms, such as TCP/IP and remote procedure call. Explicit structure determines the logical connectivity between processes, masking differences in communication mechanisms. HPC supports one-to-one, parallel channel, and many-to-many (multicasting) connectivity. Efficient computation of end-to-end connectivity from the communication structure is a challenging problem, and a third-party connection facility is needed to implement dynamic reconfiguration when the logical connectivity changes.

Explicit structure also supports grouping and nesting of processes. HPC uses this process structure to define meaningful protection domains. Access control is structured (and the basic HPC facilities may be extended) using the same powerful tools used to define communication patterns. HPC provides escapes from the strict hierarchy for direct communication between any two programs, enabling transparent access to global services. These escapes are carefully controlled to prevent interference and to preserve the appearance of a strict hierarchy.

This work is also a rare case study in consistency control for non-trivial, highly-available services in a distributed environment. Since HPC abstraction and composition operations must be available during network partitions, basic structural constraints can be violated when separate partitions are merged. By exhaustive case analysis, all possible merge inconsistencies that could arise in HPC have been identified and it is shown how each inconsistency can be either avoided, automatically reconciled by the system, or reported to the user for application-specific reconciliation.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1. Introduction

A thesis with a long title such as *Hierarchical Process Composition and the Dynamic Maintenance of Structure in the Distributed Environment* is either very wide ranging or narrowly focused. This thesis is a study in depth, rather than breadth, of one method for organizing large, distributed programs. The emphasis is on structural or architectural issues in distributed software, especially interactions involving change, protection, and distribution. The major contribution or novelty of this work is found not in the organization method, but in the detailed consideration of how its features interact with each other and with the environment.

This introduction addresses the questions "what", "how", and "what's different". Section 1.1 discusses the kind of programs under consideration, the environment in which they run, and what we want to do with them. Section 1.2 describes the basic features of the organizational method (HPC), and Section 1.3 describes how this method differs from a variety of related systems.

1.1. The Problem Area

This thesis studies the structural implications of one method for structuring large, distributed programs. As neither the method nor the problem it addresses are relevant for all programs, here we describe the computational environment, the typical program, and the type of operations on programs we are addressing.

1.1.1. The Structure of Target Applications

The number of large, distributed applications is gradually increasing while the set of tools for structuring and managing them is not. Here are selected, real applications and some of their relevant structural properties.

Complex production automation software is becoming common in industrial plants [Dou84], [Ran83], [FHH87]. It is characterized by a hierarchical structure with well defined communication patterns between nodes, great heterogeneity of computer-controlled processing and handling stations, and a significant degree of dynamic reprogramming of stations on the fly. There is also less frequent reconfiguration to add and remove stations and to modify their groupings; during a reconfiguration most of the plant is in continuous operation. The upper levels of the production automation hierarchy interact with independently maintained and administered suites of management and engineering software.

Many of these characteristics are present, but less pronounced, in process control software for industrial applications and scientific instrumentation. For example, GPIB instruments are remotely programmed for each experiment or test within an experiment [IEE75], [LoA78], [MuJ78], while space probes may be reprogrammed in flight once or twice in a year [LMW86]. A scientific experiment might be organized into a small hierarchy with several instruments controlled by laboratory minicomputers at the bottom, and the lab minis, a database machine, a numeric processor and a display workstation at the next level.

Collaborative network services are another class of distributed application. What a client perceives as a unified logical service may be implemented as a dynamically varying collection of peer servers. The DARPA Internet domain name service is a well developed example of a collaborative service [Moc87b], [Moc87a]. The actual servers are independently administered, and none of them provides complete service. Instead they provide service only for pieces of the domain name space and referrals to other servers with adjacent parts of the name space. The domain name service can be dynamically reconfigured to change the partition of the name space among servers, and to change the degree of replication of given servers, without affecting the service provided. As each server is managed autonomously, configuration is a cooperative and incremental process. The Xerox Clearinghouse and Grapevine services are other, earlier, examples of this kind of collaboration [SBN83].

A significant class of distributed application emphasizes robustness and the ability to survive failures by recognizing failures and taking corrective action. Such actions often change the operations of the application. This is a more general approach than conventional fault-tolerance in which failures are masked and ide ly have no effect on the application. A classic robust application is distributed network routing as implemented the ARPANET [MFR78], [MRR80]. The individual packet switching nodes collaborate to recompute the best rout om one node to another as nodes and links fail and recover, and as links become more or less congested. (Here we are considering the internal routing algorithm as the application, rather than the service provided by the ARPANET.) This example differs from the domain name service by being centrally adr inistered, but shares the property of having long-lived and clearly defined communication patterns between cooperating peers.

Work in distributed problem solving has stimulated a wide range of relevant program structures[1]. Contract net systems are a good example, having been used for distributed tracking of vehicles within a geographic area [Smi78], solving heuristic search problems [Smi80], and factory automation [ShW88]. A contract net system is a dynamically self-organizing program for allocating work to a set of processing nodes. A node breaks a complex task into several subtasks that can be processed concurrently, and requests (usually all) other nodes to bid for a contract on a subtask. The requesting node evaluates the bids received and awards one or more contracts. A contract net is the graph of contracts that specify how the responsibility for completing a top-level task has been broken up and distributed among nodes. Contract net systems smoothly adapt to varying loads, automatically migrating new processing to the nodes with idle capacity. They organize cooperative activity among autonomous nodes. Robustness is provided by periodically reissuing requests for bids if a task is uncompleted for whatever reason.

There are also obvious military applications under the broad heading of command and control systems. A combination of satellites, airborne platforms, microwave links, and ground mobile packet radio networks, operating under conditions that encourage frequent loss and reconfiguration, provide the data communication layer for military command and control activities. Applications, as well as the underlying communication networks, must support reconfiguration and continuous operation.

To summarize, the typical application under consideration has most or all of the following characteristics:

- It displays a *hierarchical structure* where a functional unit at one level is implemented as a collection of cooperating units at the next level down.
- It has long-lived, *well-defined communication patterns* defining the interactions between the siblings at a given level.
- Its components are *loosely coupled*, able to do significant work without an immediate response from a neighbor. They are often organized as functional *peers*, rather than master/slave or client/server.

[1] An excellent introductory survey is [Dec87].

- Its components represent *active* computational elements, like processes or tasks, rather than passive objects, like code modules or data files.
- Parts of it are *managed autonomously* so that both computation and administration are distributed.
- It is *robust* and *adaptive* to the changing conditions of a distributed environment.

1.1.2. Dynamic Maintenance of Structure

Distributed applications emphasize change. Adaptive programs are expected to change not only their behavior, but their internal structure, in response to new demands and environmental conditions. A long-lived application may be expected to run continuously for longer than any given host machine or software version will survive. Failure, migration, reconfiguration, and changing requirements all may force changes within an application.

The interactions between applications are subject to change as well. Stable distributed services usually have dynamically changing clients. In a complete distributed processing system, complex multiprocess programs are manipulated even at the highest level, where entire applications (i.e., jobs) are introduced and removed over the lifetime of the system.

This emphasis on change is a departure from the conventional environment, where the pieces of an application and their relationships are specified statically and relatively easily. Controlling and constraining change is a major technical challenge that confronts distributed programming. A static (or compiled) description of an application's structure, its distribution across host machines, and its interactions with other applications is insufficient. A framework for structuring large, distributed programs must also provide operations that modify application structure during execution.

- In general, *maintenance* encompasses functions such as replacement of failed components, compensation for partitioning, upgrading components to more recent software, and reconfiguring an application to handle more or fewer tasks.
- The combination of autonomously administered components and dynamic change requires runtime *access control* to ensure that only authorized pieces of an application are examined or modified. The same restrictions are necessary to ensure that different applications do not interfere with one another.
- Performance and engineering issues dictate consideration of *migration* or relocation of an application's pieces to accomplish load balancing, exploit locality, compensate for loss or gain of physical resources, and so forth.
- Every complex application will require some form of structural *debugging* to supplement conventional debugging of individual components. Debugging may take the form of examining the application's current structure, monitoring the communication between components, and making temporary alterations. When bugs are found, the maintenance procedures allow the necessary repairs.

1.1.3. The Distributed Environment

There is a continuum between the extremes of centralized and distributed computing and no clear boundary can be drawn between the two. Indeed, much work has been invested in supporting the centralized behavior to which programmers (and paying customers) are accustomed using ever more widely distributed hardware. Remote procedure calls, network file systems, atomic actions, and even network-wide shared memory are (not always) successful attempts to mask the distribution of the system, or provide network transparency.

However, we are interested in the distributed extreme of the continuum for several reasons.

- Extremely distributed systems have several clear, intrinsic characteristics. Primarily, their processors (sites, hosts) are *asynchronous*, and subject to *independent failure*. These properties have significant impact on the software that must run on them.
- Distributed systems are often (but not intrinsically) divided into *autonomous* regions for administrative reasons, and composed of *heterogenous* elements. Many systems can be temporarily *partitioned* into subsystems able to communicate internally but isolated from one another by failures.
- Beyond a certain physical size, it is no longer reasonable, even if possible, to mask distribution Instead, distribution should be made explicit in order to *exploit locality*, both physically and functionally.
- There are definite physical and engineering limits to masking distribution. For example, the speed of light is already a significant factor in the latency of satellite assisted communication. The availability of services that depend on simultaneous access to all copies of heavily replicated data decreases rapidly with increase in replication. Software that accounts for distribution explicitly may *scale*, while systems that depend on a centralized environment will not.
- Extremely distributed systems can not make the closed world assumption common to centralized systems, where all the interacting pieces (programs, modules, applications, client, servers) can be described all at once and in one place. Instead, they must assume an *open system*, allowing new pieces to be added to the existing framework.
- Distributed systems must admit *dynamic structure*, so that pieces can be added to and removed from the system at different times and places.

The ARPANET and SATNET [JBH78] wide-area networks and their attached hosts exemplify distributed environments, while packet radio networks [JuT87], [KGB78] and the NASA Deep Space Network [Yue83] represent some extreme cases. However, the characteristics at the end of the continuum describe distribution independently of hardware issues like relative speed, geography, and cost. The blackboard and contract net program structures used in some artificial intelligence work yield extremely distributed software by our criteria, even when implemented on centralized hardware. Therefore, we will treat the distributed environment as a programming environment, no matter where it is found, rather than a physical environment.

1.1.4. Goals

This thesis has three general goals:

- Develop a structural representation for target applications.

The representation must be adequate to describe any snapshot of a target application. It must allow for the application features described in Section 1.1.1. This will make the transition from structured design to implemented application direct, and therefore fast and easy.

- Provide operations to manipulate structural representations during execution.

These operations must provide sufficient mechanism to implement the dynamic maintenance features of Section 1.1.2. It must not be possible to create illegal representations from legal ones (soundness), and it should be possible to create any legal representation (completeness). There must also be a practical method for implementing the operations and making them available to application designers and managers.

- Identify specific environmental influences on application structure and management.

Independent failure, asynchrony, and autonomy have pervasive effects on the organization of an application. These effects will be reflected in many extremely distributed applications, whether they use our particular representation or not, and we seek to identify them. In the context of our representation, the environment oftens limit our ability to express or guarantee desirable properties. In other cases, it suggests a unification and simplification of several features.

Many interesting topics in distributed systems have been deliberately omitted from discussion. This thesis does not do many things. It does not formally define processes or active computation. It does not develop a formal model of concurrency. It does not provide a new design methodology. It does not promote new programming language concepts. It does not provide a performance model. It does not schedule processes. It does not develop new communication protocols or network architectures. It does not manage resources. It does not mask failures. It does not serialize application operations. Most of these issues are independent of *any* form of program structuring and represent services that can be provided by host facilities beneath the system to be described or by utility applications above it.

1.1.5. Thesis Outline

The remainder of this Introduction sketches the HPC approach to structuring applications, based of process abstraction and explicit composition, and compares it to related work. Chapter 2 introduces three of the four exploratory themes of this thesis: protection and control structure, communication structure, and non-hierarchical structure, and illustrates the HPC operations for run-time reconfiguration. The interactions among these features and between them and the environment are noted throughout the following four Chapters.

The HPC protection system defines what an agent is permitted to examine or change. Chapter 3 shows how we exploit rich and explicit process structure to define meaningful domains of protection, and how control is configured using the same powerful tools as communication. Some major benefits from this unique protection system are direct association of protection and management, arbitrary user-defined access control policies, and a simple mechanism for extending or modifying the built-in HPC system facilities.

In Chapter 4, we focus on interprocess communication (IPC). Starting with simple one-to-one communication patterns, HPC incorporates multiple parallel channels and arbitrary many-to-many patterns. These complex interactions are all expressed structurally, instead of using addressing or other properties of specific IPC mechanisms. HPC supports heterogeneous IPC mechanisms with differing behaviors, while presenting a single mechanism for the configuration of communicating processes. This prompts a division of communication functions into logical configuration, transport implementation, and actual communication.

A purely functional approach to composition involving strict trees and explicit composition is impractical. In particular, access to global services is clumsy and potentially dangerous. Chapter 5 demonstrates the clash between transparent abstractions and purely functional compositions. HPC resolves the clash by allowing direct non-local communication between any two points in the hierarchy, while preserving the appearance of a strict tree.

The target environment is subject to partition, and HPC permits highly available applications that continue to run with reduced resources while partitioned. Because process structure may be freely modified during partition, inconsistencies can be discovered upon merge. The fourth exploratory theme is the reintegration of applications that have been modified inconsistently during partition. Chapter 6 identifies all possible HPC merge inconsistencies, and shows how they are either avoided, automatically reconciled while preserving the pre-merge behavior, or reported to the user with tools for application-specific reconciliation. The techniques we use for avoiding inconsistencies are not specific to HPC, and provide useful lessons for building other, highly available, reconfigurable systems.

Chapter 7 reports the prototype HPC implementation and early experiences with it.

Following our conclusions in Chapter 8, we present a formal description of HPC structure and the operations on it in Appendix A. Soundness and completeness results are related to HPC structure considered as a formal system. Based on our experiences, we suggest investigation of new laws of distribution and composition for strictly hierarchical formal systems such as CSP and CCS, in part to provide sharing that those systems do not support.

1.2. Hierarchical Process Composition

The process abstraction has been used successfully as a tool to structure complex systems since the late 1960's. The THE [Dij68] and RC4000 [Bri69, Bri73] operating systems with their layers of cooperating processes are important early examples of such *process structuring* [HoR73]. When considering how to organize a target application, consisting of loosely-coupled, active peer elements with well-defined communication patterns, process structuring should come to mind immediately as an appropriate choice.

Brian Randell has emphasized the additional structuring principles of abstraction and composition in his work on reliable software.

> Thus the sorts of structuring that we have discussed so far can be described as structuring within a single level of abstraction, or *horizontal structuring*. ... In choosing to identify a set of levels of abstraction ... and to define their interrelationships one is once again imposing a structure on a system, but this is a rather different form of structure which we will refer to as *vertical structuring*. Thus vertical structurings describe how components are constructed, whereas horizontal structurings describe how components interact. [Ran79]

He also described the degree to which the logical structure of an application intended by the designer is supported and enforced by the underlying system as the degree of *actual* as opposed to *conceptual* structure.

We are motivated by distribution rather than reliability, but the concepts of process structuring, abstraction, and composition are the basis for Hierarchical Process Composition (HPC). In Randell's terminology, we will put actual structure into distributed programs, with explicit vertical and horizontal structuring.

Our focus is on process structuring and how complex applications can be built from smaller ones, and not on the internal behaviors of individual processes. For this reason, the definition of *process* is not critical to HPC. Any favorite definition (finite state automata, infinite sequences of primitive events, or state vectors plus threads of control) may be used. The only things we need to know about any particular process are its name and the interfaces where it may interact with other processes. These external properties define the process *abstraction*.

Horizontal structure defines a graph of processes and the interactions between them. We often call the pattern of communication in this graph the *composition* of processes, because it defines the behavior of the overall structure as a function of the behaviors of its components. Vertical structure groups related processes together. By extending the process abstraction to groups of processes, vertical structure provides the hierarchical structure typical of a target application. By making the representation of horizontal and vertical structure explicit, maintaining it during execution, and forcing applications to reflect their representations, actual structure can be enforced using the mechanisms provided for dynamic maintenance.

1.2.1. Explicit Communication Patterns

To gain the greatest actual horizontal structure, we must make interfaces and the bindings between them as explicit and visible as the associated processes. We will consider only communication between explicitly identified partners. Each pair of partners interacts via a communication *medium*, whose characteristics do not concern us here. (Examples are TCP/IP connections, semaphores, shared files, remote procedure call bindings, wires, and UNIX[2] pipes.)

A process has a fixed set of communication *interfaces*, one for each potential partner, that may be thought of as endpoints or sockets for communication media. A process's interfaces are distinguished according to the role played by the partner: kernel port, logging, auditing, standard input, standard output, mailbox server.[3] A *connection* is an instance of a communication medium joining two interfaces. We identify connections by the interfaces at their ends.

We will now consider two small examples. Take a typical UNIX pipeline of processes *A*, *B*, and *C*. Every UNIX process has a conventional abstraction: three interfaces (file descriptors) for standard input, standard output, and standard error. In a pipeline, the output interface of one process shares a UNIX pipe with the input interface of the next process in line. One process views the pipe as a sink, while the other views it as a source. The remaining interfaces are connected to the terminal device by default. In Figure 1.1 this pipeline (without the terminal device) is shown in the notation that will be used throughout. Processes are drawn as shaded rectangles; interfaces as small tabs on processes; and connections as heavy lines.

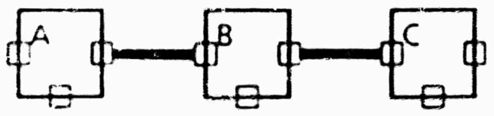

Figure 1.1. Simple Pipeline

The example illustrated in Figure 1.2 uses remote procedure call (RPC) bindings rather than UNIX pipes as the basic communication media. Processes *FileServer* and *NameServer* each has an RPC interface, corresponding to its external entries, to provide its service. Process *Client* has two interfaces, corresponding to its stubs, one to obtain file service and the other to obtain name service. Connections between interfaces indicate RPC bindings. It is important that *Client*'s interfaces are distinguished so that an appropriate server can be bound to each set of stubs.

[2] UNIX is a registered trademark of AT&T.

[3] Consider channel identifiers in [Bla83]

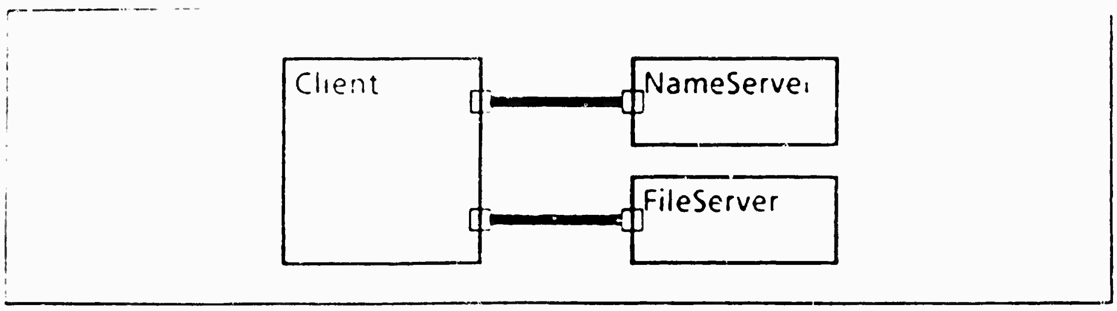

Figure 1.2. Simple Client and Server

We are not so cavalier about the semantics of communication media as might appear from this introduction. Later Chapters explore the integration of multiplexing, multicasting, and bundling paths into this initially one-to-one model of communication, the implications of requiring explicit partners, the distinction between physical and logical media, and the degree to which several common interprocess communication mechanisms fit the HPC model.

1.2.2. Nested Groups of Processes

When two multiprocess programs are connected, the boundary between them is lost in the composition. To retain the abstract grouping of related processes, we must incorporate vertical structure. An HPC *object* is a named, active entity with distinct interfaces, just like a process. However, an object is implemented by an explicit composition of processes that can be described and manipulated in the HPC system, while processes are implemented by some primitive behavior that can not. The boundary between the external abstraction and the internal composition of an object is a *shell*.

Objects obviously capture nesting or vertical structure at one level, and it is natural to extend the process abstraction by allowing an object anywhere a process could be used. This leads immediately to a hierarchy or tree of objects rather than a single level of clustering or grouping. The leaves of the tree are real processes running on some machine. All the nodes of the tree can be treated like real processes, but the internal nodes are collections of abstract processes, some of which may be real processes and some of which may themselves be collections. Near the top of the tree we find abstractions dealing with activities of broad scope and complexity. Near the bottom are abstractions with simple behavior and limited complexity.

A UNIX pipeline can itself be used as a component in a larger pipeline. The command `(A | B) | (C | D)` defines a pipeline of two components, each of which is a pipeline of two components. However, in UNIX this logical nesting is completely lost by the time the command is implemented. In HPC we represent the nesting explicitly as shown in Figure 1.3. A shell is drawn as a rectangle surrounding the contents of the corresponding object.

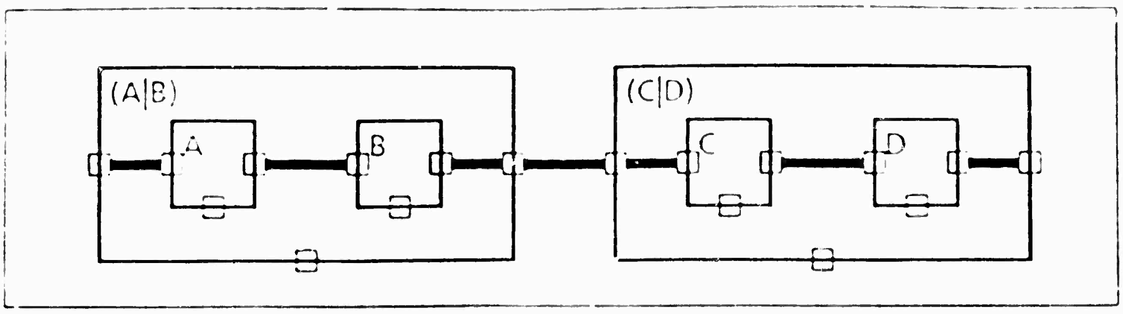

Figure 1.3. Pipeline of Pipelines

By encapsulating our complex client and server programs as objects we can readily separate their internal process structures from the interactions between them. Figure 1.4 shows how the *Client* and *NameServer* of Figure 1.2 can be expanded internally to more complex structures without affecting the interactions between them.

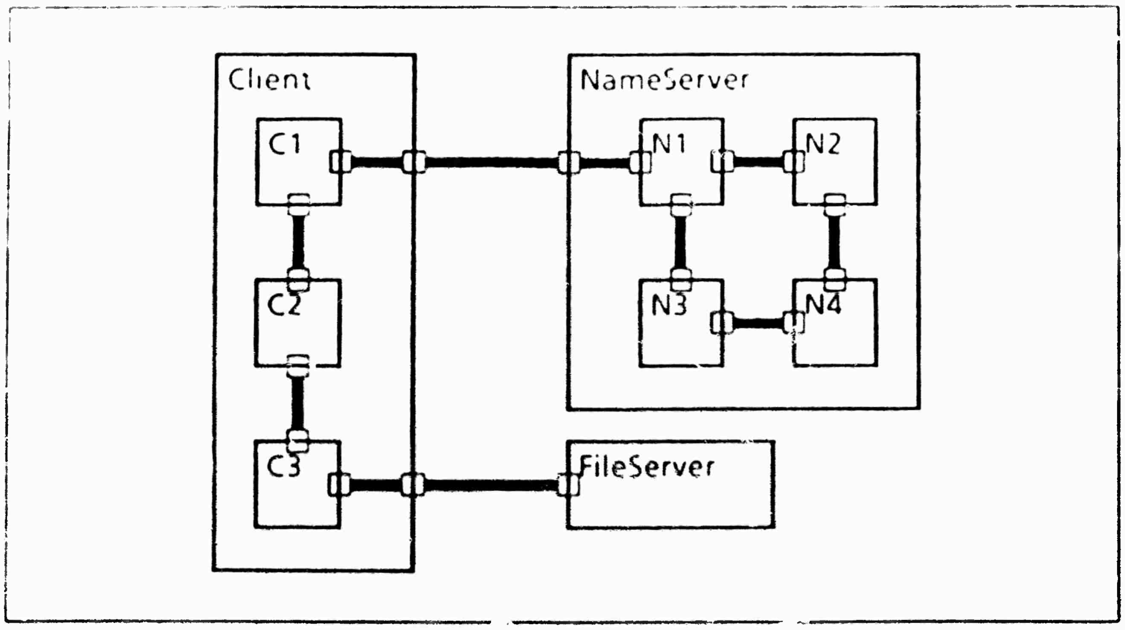

Figure 1.4. Multiprocess Client and Server

Hierarchical organization is a natur[illegible]me of applying the principle of abstraction generously. It is a good method for implementing complex [illegible], be[illegible]use it closely approximates the structure designers actually use in creating their applications (Sect[illegible]). A[illegible]e there other, perhaps better, ways to organize complex applications than to force everything into hie[illegible]es? Our response is "probably not." To quote from Simon:

> The fact, then, that many complex systems h[illegible] a nearly decomposable, hierarchic structure is a major facilitating factor enabling us to understand, to describe, an[illegible]ven to see such systems and their parts. Or perhaps the proposition should be put the other way round. If there [illegible]e important systems in the world which are complex without being hierarchic, they may to a consid[illegible]able exte[illegible] escape our observation and our understanding. Analysis of their behavior would involve such de[illegible]iled knowledge and calculation of the interactions of the elementary parts that it would be beyond our capacities [illegible] memory or computation. [Sim62]

However, strict hierarchies can not realistically express the kind of sharing and access to global resources needed in a real system. Unless the mathematical elegance of strict functional composition is the sole criterion, this limitation must be overcome. This issue is explored in detail in subsequent Chapters.

1.2.3. Active versus Passive Hierarchies

Even after a decision to use hierarchical grouping as a method for organizing complex software, there remains a choice between active and passive hierarchies. In an *active hierarchy* the internal nodes in the tree are processes. All interactions with a subtree are actually interactions with the process at its root. In a *passive hierarchy* the internal nodes of the tree are abstractions. All processes are at the leaves of the tree. In both cases we assume that, to obtain the benefits of the hierarchical discipline, a node can be connected only to its parent, its children and its immediate siblings.

To some, active hierarchies may seem like the natural choice. No unfamiliar abstract objects are introduced, it is clear where control of a process group resides, and the process hierarchies supported by most existing operating systems (e.g., UNIX) are of this type. However, they are an inadequate solution to the problems we are trying to solve. Let us examine the obstacles active hierarchies would present.

First of all, an active hierarchy is insufficiently abstract. We really do want to introduce those unfamiliar abstract objects. It is very important to distinguish a complex application from the processes that happen to be implementing it at the moment. There must be a name or placeholder for a subtree independent of its root process, or else the root process can not be transparently replaced by another.

Second, active hierarchies do not extend the process abstraction well. A subtree can not cleanly replace an arbitrary process. Replacing a leaf with a subtree, or vice versa, is clean, but replacing an internal node (subtree root) with a non-trivial subtree *always* destroys the relation its children had with the internal node, and generally destroys the sibling relation among them as well. Permissible connections depend on the sibling relation, so this is a serious problem.

Third, a single process at the root of a subtree represents an unacceptable potential for single-point control failure. Redundant control is critical for some distributed applications. There must be a way to spread control responsibilities for a subtree among several processes, or at least ensure an automatic promotion to the root for alternate processes in the event of the failure of the current root.

These obstacles can be overcome by the introduction of something akin to our abstract object, so that active hierarchies are now trees of objects, which might have a single process or an active hierarchy within them. But now the active hierarchy has become a passive hierarchy! Provision for redundant control requires even further extensions.

1.2.4. Dynamic Process Structure

A static snapshot of an application is intrinsically simpler than a specification that describes how the application is to adapt and evolve over time. In an open system, future components (and the future policies governing further changes) may not even exist when a snapshot is taken, so a fully general specification must be open-ended in some sense; it can not encompass all future configurations of the application. For this reason, we use a procedural, rather than declarative, description of change, and define the mechanisms by which an application may be modified, rather than the policies governing the appropriate modifications. In fact, we will have no completely static representations. All application structure is described dynamically; even a nominally static structure must be built incrementally from an empty structure using the transformation operations.

There are six basic operations on process structure in HPC. These are create and destroy process, create and destroy connection, and create and destroy shell. There are no general tree editing operations such as move subtree, only operations which create new structure, have generally local effects, and their inverses. Creating and destroying processes are basic operations in most operating systems, but the other operations are usually limited or unavailable.

There is also an operation to examine the process structure at any moment. This gives an immediate advantage over existing systems for inspecting and manipulating multiprocess programs. Most operating systems can provide a list of the currently active processes, but not a list of which processes are interacting with which others. By examining the process structure, we can tell exactly how processes are interacting and what the logical significance of each interaction is.

A (distributed) service maintains a database of the current dynamic process structure, and translates the basic HPC operations into the necessary low-level host operations on processes and so forth. This HPC service runs as a user program on top of conventional host operation systems, and provides an abstract interface for application managers. Most operations on HPC process structure require only local database manipulations, and involve no physical resources from the underlying hosts.

1.3. Related work

In some ways, HPC represents the last of a long period of loosely-coupled distributed systems work at the University of Rochester. The well-known Rochester Intelligent Gateway [BFL76] distributed operating system and PLITS [Fel79] distributed programming language established a strong departmental interest in an asynchronous, message-passing model of interaction. Two subsequent projects, Activities [EFH82] and Super [Ary81] are specially related to HPC.

The Activities work is the primary starting point for HPC. The activity model provides a tool to describe the relationships between objects involved in the execution of a shared, distributed task. A single object may participate in many different activities and a single activity may be made up of numerous subactivities. At the language level, tags are used to identify the activity affiliation of data and messages [Hel84]. HPC began as an attempt to define, in detail, operating system support for the activity model. It quickly diverged, although previous work on activities had important influence throughout.

Super is an exploration of communication via broadcast source-addressed messages, and of how language support of programs using such a medium could be provided, that parallels HPC in several ways. The Super programming language provided nested groups of processes, together with distinguished processes to control and manage such groups. Both of these constructs are primitive in HPC structural representations. Super also requires the concept of a secretary process (communication filter); Such filters are frequently convenient in HPC, but are neither critical nor built in to the system. We indicate these parallels as evidence for the universal nature of the structures, at least in a loosely coupled environment, as Super had no direct influence on HPC.

1.3.1. CONIC

Beyond a doubt, the related work most closely related to HPC is the CONIC toolkit, developed independently of, and earlier than, HPC at University College London [KMS87], [KrM85]. Like HPC, CONIC provides for explicit communication interfaces and bindings, a process abstraction generalized to nested groups of processes, and fully dynamic structure. The CONIC system implementation is far more developed than the HPC system prototype and has been applied to several industrial applications.

The major original contributions of HPC are easiest to evaluate by comparison to CONIC. First, CONIC as currently implemented provides a single native communication mechanism, while HPC was designed to use a variety of (heterogeneous) mechanisms. As a result, HPC has facilities to "type-check" communication paths to ensure each logical path can be implemented. More significantly, HPC provides for explicit expression of multiplexing, multicasting, and bundling of multiple communication paths as part of the horizontal structure to capture the richness of communication media.

CONIC has no provision for protection or domains of autonomous management. Most systems that use a general protection system such as access control lists or capabilities, do so because they have no obvious structure to exploit. HPC uses rich vertical structure in the definition of protection domains that provide common management for related objects. In addition, the "controls" relation, which is as important as the "communicates with" relation, is manifest in HPC. Control behavior is subject to the same principles of abstraction and composition as other process interactions.

While both systems concentrate on hierarchies of processes, HPC also allows exceptions to a strict tree. This permits more natural access to global services than is possible in a tree. The more general graph structures resulting from exceptions have to carefully disciplined to preserve the behavior of a strict hierarchy, while still allowing access between arbitrary points in the tree; it would not be practical in HPC without exploiting the protection system.

CONIC and HPC had somewhat different motivations that account for some less well defined differences. HPC has an emphasis on extremely distributed systems. This led to features, e.g. continued operation during partition and reporting end-to-end connectivity without violating abstraction boundaries, that are not as well developed in CONIC. As the HPC system was not intended as a stand-alone system nor as a complete one, its relation to application processes and to host operating systems is more precisely defined than in CONIC. In particular, manipulations of rich, abstract application structure are completely distinguished from the primitive operations on sparse, native processes and communication media. This precise abstract structure, in turn, has suggested specific research into additional distributive laws for formal systems such Hoare's CSP [Hoa85].

1.3.2. Task Forces

Since the late 1970's, several lines of operating system research have explored an explicit form of structuring for multiple processes often known as *task forces*. A task force usually consists of a variable number of processes performing the same or similar functions.

One prominent line of research stresses the independence of address space and thread of control, and the resulting efficiencies due to shared memory communication and faster context switches between processes using the same address space. The related Thoth [Che82], Verex [Loc79], and V kernel [Che84] systems, and the unrelated

Mach [YTR87], [ABB86] system are examples of this development. While the grouping of processes into task forces (Thoth: tasks into teams, Mach: threads into tasks) is very well defined in these systems, there is little or no support for further structuring of processes in the same task force, or of multiple task forces. Communication is based on promiscuous broadcast within a group or other mechanisms (ports, links, addresses) that can not be treated as explicit, visible bindings.

A second line of development emphasizes a object-invocation model of interaction, where multiple active processes may service concurrent invocations of a given object. Three well-known examples are Argus [LiS83] Eden [ABL85], and Clouds [LeW85]. Communication interfaces in these systems are defined and controlled very clearly on the receiving side of an invocation, but the binding between calling and called objects is left implicit in the pattern of invocations during runtime. Most analyses (e.g., for deadlock freedom, or for debugging) of applications run on such systems require the "calls" or "depends" relation between objects, which again suggests that the horizontal structure should be manifest in the structure, as in HPC, and not inferred from the dynamic behavior.

Vertical process structure is limited to the single level of process clustering within objects. However, both Argus and Clouds provide additional structuring in the form of transactions that define apparently atomic activities that may involve processes within several objects. Even ignoring the aspect of atomicity, HPC has no comparable facility for defining activities that involve several applications, perhaps overlapping with other such activities, only for grouping applications into a larger one.[4]

Two important systems based on capability-controlled access and message passing were implemented as part of the CM* multiprocessor project ([SFS77], [SBL77]). These are StarOS [JCD79], and Medusa [Ous81], and both systems allow the creation of distributed task forces of cooperating processes. StarOS focuses on ease of use and a general capability mechanism, while Medusa stresses the effect of distributed hardware on system software (Section 8.2.1 [Ous81]). Medusa, more than any of the other task force system, addresses the structural issues central to HPC.

Each Medusa process has a private capability list, the processes in a task force share a list, and every process has access to a list of global utility capabilities. Horizontal structure in Medusa is explicitly controlled by these lists, which are distinct from the processes that may access them. Each slot in a list can be treated as an abstract interface, where the capability in the slot specifies the implemention. This definition of explicit interfaces is so clean and comprehensive that the complete state of a process, including its memory pages and access to secondary storage, is accessible through its capability lists. One pleasant result is that one process can take over completely for another in the same task force either temporarily ("buddy" exception handling) or permanently. Dynamic load balancing and system reconfiguration is possible by replacing the capabilities for overloaded or failed processes on the fly. HPC can only approximate this clean replacement, because process state is a primitive feature outside the HPC structural system.

In Medusa, unlike StarOS, the vertical structuring of processes into task forces is maintained during execution, available for debugging and monitoring. However, Medusa is a one level system. There is no facility for

[4] This, despite HPC's direct roots in Rochester Activities.

grouping task forces into larger applications, and utility task forces are treated somewhat specially by the system. The same directness applies to communication; while capabilities can be replaced, they can not be chained, indirected, or sent in messages. One result is that a task force must explicitly provide for each of its users, there is no way to export or otherwise transmit access to a task force. HPC provides much more powerful organizing tools in this area.

1.3.3. Software Design Tools

Software design tools emphasize methodical development, clean abstraction, and reuseable modules. This emphasis often encourages (or enforces) software architectures that have a great deal in common with loosely coupled, distributed applications. In particular, they invariably provide nested abstractions with explicit interfaces and intermodule bindings. When a design tool provides active entities as a basic component, the possible structures are much like static HPC structures.

The SARA system is one such tool that has been used to design, model, and simulate both sequential and concurrent software [EFR86]. SARA distinguishes vertical and horizontal structure to the extent that different languages are used to describe them [PeB79], but goes beyond structuring (syntax) to include a third language to describe the behavior (semantics) of an application. Intentionally, there is no comparable feature in HPC.

There are many other software design tools with some relevance to HPC, but we will mention in passing only SADT [Ros85], and DREAM [Rid81]. The relationship of HPC to SARA, along with these other tools, is more complementary than competitive. While HPC has no semantic component, and the design tools do not allow for dynamic structure (failure, reconfiguration, etc), the methodology and design-time support provided by the tools can be usefully matched by the run-time support provided by HPC.

This match has been realized by the STILE/GEM combination [SBW87]. STILE is a software design system of the type discussed here, while GEM is a real time operating system for robotic applications. The GEM multiprocessors are physically close together, the software environment is quite loosely coupled, and STILE/GEM allows for direct run-time support of design abstractions, explicit distribution and (limited) dynamic reconfiguration. Because GEM was intended for specific applications and hardware environments, it does not address issues like protection, partition, and dynamic process creation that are addressed by HPC. Like CONIC, GEM provides a complete, native, run-time system, while HPC does not.

1.3.4. Programming Languages

Several programming languages support nested processes, in either passive or active hierarchies. A number of these languages are related to Hoare's CSP [Hoa85], either directly or by parallel development; ECSP [BRT84], Occam [TaW82], Planet [CrE84], and Platon [StS75] are four examples. Because these languages express structure directly in program syntax, they have very limited ability to express either dynamic or persistent structure. ECSP, for example, allows for dynamic reconfiguration, but all potential module bindings must be manifest in the original program. Process lifetimes are limited to a strict fork-join discipline, so processes can not be detached to run on their own, nor can new ones be added to a group once it has been created. These limitations are intrinsic to any system that treats each application (program) as a closed system. HPC manages applications as an open collection of persistent data structures.

While interfaces and bindings between siblings are fairly well defined in CSP-like languages, the bindings between processes at greater distances in the tree are extended by scoping and visibility rules of the language. These rules limit the possible patterns of communication but do not make the specific patterns explicit. (In ECSP, the specific patterns are not even decidable in general.) A more serious problem is that each process must name its communication partner(s) directly to create a binding. This limits the use of abstraction, because a process must have information about arbitrarily distant structure. All HPC connections are strictly local, which allows rigorous information hiding, and bindings are computed incrementally on the basis of possibly many connections.

The PRONET [LeM82] programming language is more attractive for extremely distributed software than the CSP-like languages. It uses a separate sublanguage (NETSLA) for explicit module interconnection that avoids the objections raised in the previous paragraph. Modules and intermodule bindings can be created and destroyed at any time, and the contents of each abstract object are managed and controlled by precisely the NETSLA code that created the abstract object. Rich communication structure, for example explicit multiplexing, multicasting, and bundling of interfaces, can be expressed in the module definition sublanguage (ALSTEN).

However, PRONET still suffers from a closed world assumption, because there is no way to introduce new types of modules into a running system nor any way to name or communicate with an independent program (e.g., contact a global service). While asynchronous creation and destruction of modules are addressed by PRONET, more general issues of decentralized control (partition and multiple agents) are not. In fact, the control and management portions of a PRONET program are implicitly single threaded and centralized, even if the primitive processes run concurrently.

There are of course very many other programming languages that allow multiple processes. The ones discussed here are most relevant to HPC and the structures it can express, and space prohibits even a simple listing of such languages. For example, DPL-82 [Eri82] is a language very different from PRONET and the CSP-like languages that also provides nested groups of processes with explicit interfaces and bindings, but the similarity adds nothing significant.

Chapter 2 Hierarchical Process Composition

2. Hierarchical Process Composition

Black boxes are good physical abstractions. They are distinguished by the sockets on their exterior panels and their behavior at those sockets. Their internal structure and state are not accessible. All we know about a black box is its name, its sockets for cables, and the signals it produces at those sockets.

Interesting ensembles of several black boxes are created by connecting pairs of sockets with cables. In such an ensemble, any cable may be replaced by an equivalent one, but generally no box or socket can be substituted for any other. Therefore, we must distinguish boxes and sockets, but need not distinguish cables.

We often wish to enclose an interconnected collection of boxes in a chassis or cabinet for convenience (or perhaps, to introduce a level of abstraction). This will hide all the internal wiring that is irrelevant to user of the overall collection. To accomplish this, we must have some sockets that pass entirely through the cabinet. On the exterior, the cabinet appears to be a black box and the sockets are used as we have already described. On the interior, the sockets must be connected to the "free" sockets of the connected black boxes.

Objects, *interfaces*, *connections*, and *shells* are the HPC analogues to black boxes, sockets, cables and cabinets. The hierarchical organization of modern electronic equipment is clear: gates composed into integrated circuits, integrated circuits and discrete components composed into circuit boards, boards composed into shared bus modules, modules composed into computers, computers composed into networks ... At each level there is an explicit composition of black boxes, which are abstractions of the next lower level. This is the way HPC uses the principles of abstraction and composition to organize complex, multiprocess programs.

However, a realistic system must be fleshed out with protection against unauthorized access or interference between applications, escapes from the strict object hierarchy when appropriate, and provision for a rich set of interprocess communication patterns. The interactions of these additional structural features are the subject of much of this dissertation. They are introduced in Section 2.1, and examined in detail in the subsequent three Chapters.

HPC, unlike black boxes, supports dynamic reconfiguration of both abstractions and compositions under program control. The system provides operations that incrementally modify process structure during execution. Section 2.2 presents the entire set of fourteen HPC operations, organized by the structural features they affect.

The first significant interaction between multiprocess abstractions and the distributed environment is a requirement for an asynchronous system interface. Section 2.3 notes the clash between synchronous operations on process structure and the asynchrony among agents and between agents and sources of failure. By stressing dynamic structure, we are led to adopt an unconventional system interface when compared to most distributed software systems.

2.1. Structural Features

2.1.1. Dual Representation

Nested HPC shells define a rooted tree of objects. A tree is often the most intuitive representation for complex applications, but a more general representation is sometimes needed. The HPC *dual graph* captures the desired escapes from a strict hierarchy and simplifies the presentation of the protection and communication structures.

The hierarchy emphasizes vertical structure, where objects are nodes and the parent-child relation defines the edges. The dual graph[5] emphasizes the horizontal structure of a system, where compositions are nodes and the abstract interfaces define the edges. Both interpretations of a small object are shown in Figure 2.1. (Hierarchies are always drawn as nested rectangles, while dual graphs are always drawn using circles.) Shells above and edges below correspond directly, while nodes in the dual graph represent *spaces* between shells in the hierarchy. Figure 2.1 successively highlights each dual node and its hierarchical equivalent.

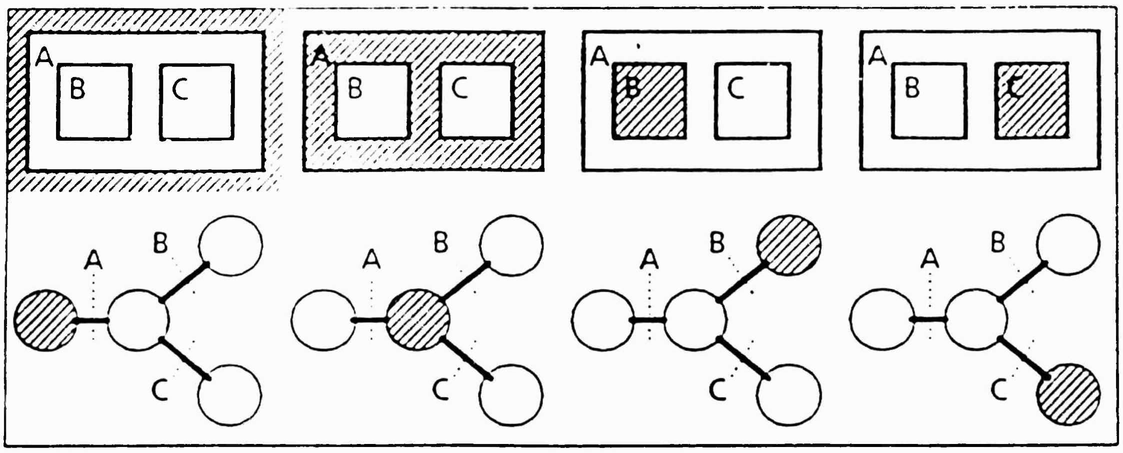

Figure 2.1. Two Representations of a Small Object

The parent and child shells in the hierarchy are the edges incident on a space in the dual graph. A space has one set of interfaces from each of these edges. In a *simple* space, these interfaces are accessed directly by a process, and in a *complex* space, they are composed by a set of HPC connections. An active hierarchy would have simple spaces throughout, while the passive HPC hierarchy has simple spaces only at the leaves.

The dual graph has a distinguished *root space* and the subtrees rooted there form a forest of hierarchical objects. These *top-level objects* are the most independent activities in the system, generally corresponding to user terminal sessions and to long-lived system services.

Every HPC hierarchy has a dual graph, but Figure 2.2 demonstrates a dual graph that can not be expressed as a hierarchy. The dual graph ignores the hierarchical orientation, which simplifies several technical definitions and allows a uniform treatment of composition in strictly tree-like and cyclic process structures.

[5] Actually, a line graph for those fussy about graph-theoretic terminology.

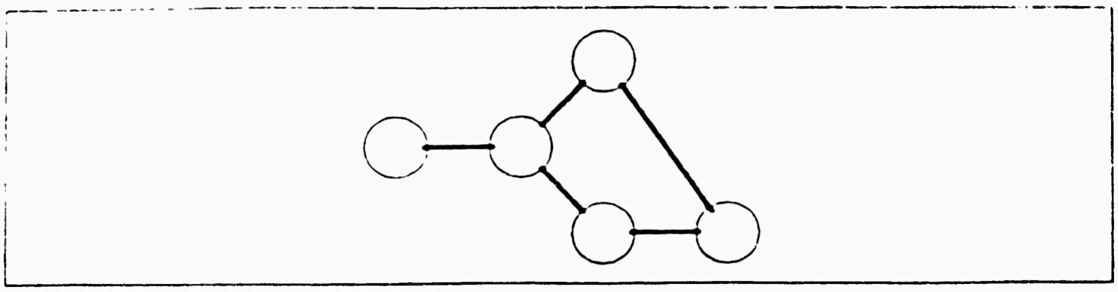

Figure 2.2. A Non-Hierarchical Structure

2.1.2. Protection and Control Structure

The basic concepts of the standard protection model are *domain* and *agent* (or *principal*) [Jon79]. A domain is a set of entities and specific operations on them. An agent is an active entity authorized to apply a domain's operations on its entities. In general, the mappings between agents and domains, and between domains and contents, may be many-to-many. That is, domains may have several agents, an operation may be in several domains, and so forth. The *protection relation* is the mapping among agents, domains, and domain contents. In conventional protection systems, the protection relation is associated either with the domains (access control lists), or with the agents (capability lists). In both cases, it is difficult to determine the inverse mapping.

HPC exploits the coherence and locality of the rich, explicit process structure to define protection domains, instead of using a conventional protection system that makes few, if any, assumptions about the structure of domains. Domains are disjoint, connected subtrees in the dual structure graph. Every space belongs to exactly one domain. In the hierarchical view, this defines a coarse hierarchy of domains on top of the hierarchy of objects. Some, but not all, shells delineate domains while all shells delineate objects. Figure 2.3 illustrates three domains. (Domain boundaries are drawn as double lines.)

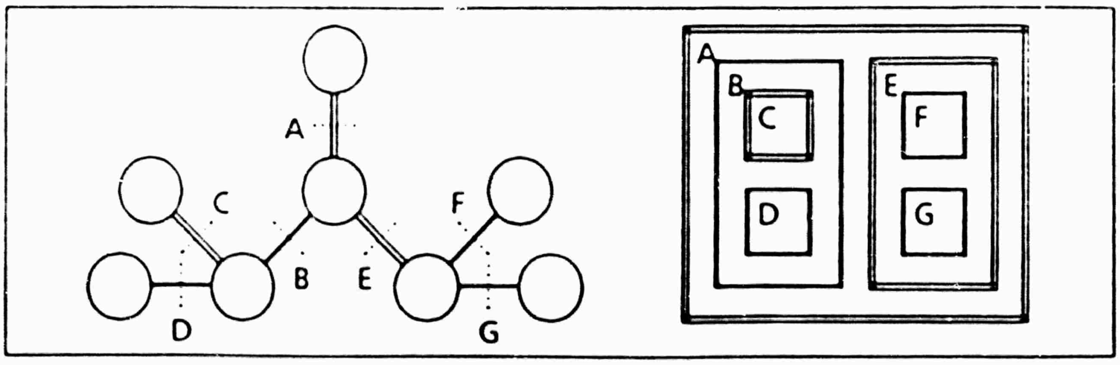

Figure 2.3. Three Adjacent Domains

All features of a space and all operations on them are in its domain, but operations requiring operands in multiple domains are not included in any domain. This omission is the fundamental protection feature, and effectively confines the effects of an operation to a single domain. An agent either has complete control over a domain or none at all. The protection mechanism provides no form of limited access such as "read only" but we will see this does not impede arbitrarily complex access policies.

Operations on HPC structure are provided through a built-in *controller* object in each domain. A controller accepts control messages from connected objects and invokes the corresponding operations on structure. It refuses to carry out operations outside its domain. (The controller does not have any physical existence, it is simply an explicit placeholder for the HPC system, and therefore can be made robust, available everywhere and whenever the HPC system is working.)

Objects connected to a controller, and thus able to exert their control over a domain, are, *de facto*, agents. Multiprocess agent objects immediately allow redundant control of robust or decentralized applications. However, partition or site failure can lead to situations where no agent is physically able to exert its control over a domain. The HPC system provides positive control to ensure every domain is under control of some agent at all times.

A small domain is shown in Figure 2.4. Domain D is a short pipeline implemented by two worker objects, A, and B. Object M manages the domain, monitoring and instructing A, and sending HPC commands to the controller C as needed. (Controller objects are drawn in dark grey.)

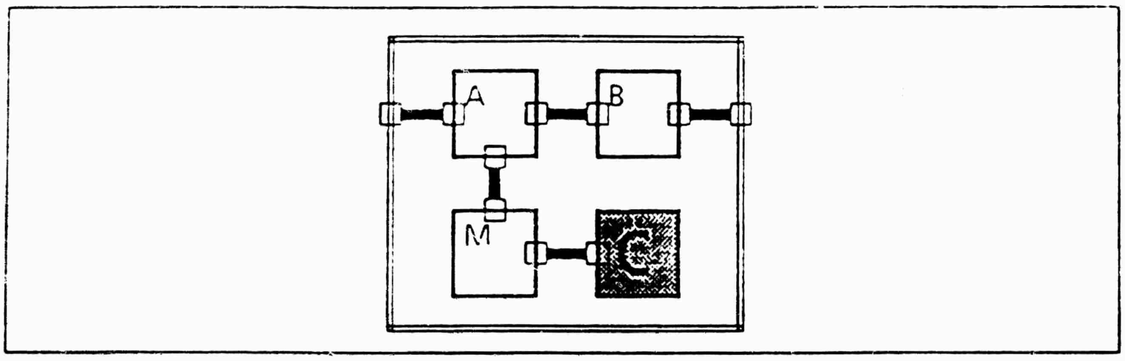

Figure 2.4. Domain with Agent

Chapter 3 discusses protectio and control structure in more detail, especially positive control and the user definition of arbitrary access policies.

2.1.3. Communication Structure

Composition is fundamental to the behavior of any system, while abstraction is merely an unavoidable convenience. That is, raw, primitive, individual components can be composed into a useful system without the benefits of further abstraction, while complex abstractions remain isolated and collectively useless without a way of specifying interactions between them and within them. Therefore, it is not surprising that communication structure, defining the possible compositions of objects, is the most complex part of HPC.

HPC supports heterogenous communication mechanisms and manipulations of multiple related communication paths in a single operation. It also extends the one-to-one communication patterns shown so far with a general many-to-many multicasting facility. These features are introduced here, with considerably more detail in Chapter 4.

Both sides of an interface can be manipulated separately. The independent parts, called *views*, are basic building blocks that can be combined in two ways. Pairs of views can be connected to form links in a communication path, and views can be assembled into tree structures to form more complex interfaces.

Simple views control individual communication media such as a UNIX pipe, a TCP/IP connection, or a remote procedure call binding. When a view is created, the appropriate mechanism is specified together with any constraints such as orientation. For example, pipes and RPC bindings are always oriented (out/in and client/server, respectively) while TCP/IP connections are generally not. Some simple view structures are:

```
[ structure: simple UNIX-stream in ]

[ structure: simple TCP/IP in-out ]

[ structure: simple Courier client ]
```

We say views are *compatible* when their specified mechanisms are the same (and their constraints are complementary). HPC requires that both views of an interface, and both views of a connection, be compatible to ensure a transport medium can be implemented for each logical communication path.

Often it is desirable to connect two objects with a single abstract medium, while allowing an implementation using more than one physical communication channel. A physical example is a cable with multiple conductors, the programming analogy is the record data structure, and distributed program examples are paired half-duplex channels, and out-of-band channels. To support this kind of grouping we use *bundle* views. A bundle has a fixed, ordered set of possibly heterogeneous *component* structures. The bundle corresponding to the conventional UNIX standard IO interface is:[6]

```
[ role: "UNIX stdio",
  structure: bundle
  [ role: "stdin",
    structure: simple UNIX-stream in
  ]
  [ role: "stdout",
    structure: simple UNIX-stream out
  ]
  [ role: "stderr",
    structure: simple UNIX-stream out
  ]
]
```

Another common requirement is for dynamically changing numbers of homogeneous media. A physical example is the trunk line[7], the programming analogy is the set data structure, and a distributed program example is a multiplexed service. This motivates the *multiplex* view. A multiplex view has a single, fixed component structure, but varying numbers of component views with that structure can be freely created and destroyed during execution. Each view represents a distinct communication channel.

[6] The *role* keyword is introduced here to clarify the example. Its full significance will be explained later.

[7] Considering virtual circuits as the media, instead of the wires.

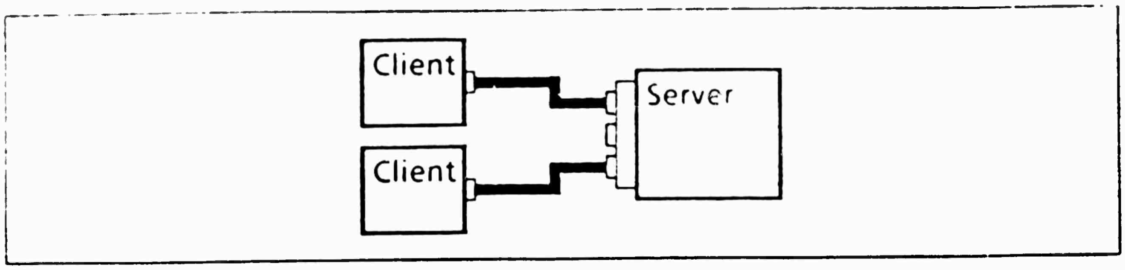

Figure 2.5. Multiplexed Server

The X Window system is a typical multiplexed service. Clients of the X Window system each have a view with this simple structure

```
{ role: "X Window client",
  structure: simple TCP/IP in-out
}
```

while the network interface of the X Window system server has this multiplex structure

```
{ role: "X Window server",
  structure: multiplex
  { role: "X Window service",
    structure: simple TCP/IP in-out
  }
}
```

The different kinds of complex views can be combined hierarchically. Here, for example, is a multiplex view with a bundle component, the bundle having two simple components:

```
{ role: "two-function server",
  structure: multiplex
  { role: "package binding",
    structure: bundle
    { role: "func-1 entry",
      structure: simple Courier server
    }
    { role: "func-2 entry",
      structure: simple Courier server
    }
  }
}
```

One-to-many and many-to-many communication patterns are a third important class of relationships. Physical examples are the bus and triple modular redundancy systems, a programming example is the FORTRAN common block, and a distributed program example is the contract bidding algorithm. HPC must demonstrate that these relationships can be expressed in abstract structure, rather than physical addressing.

The *multicast* view is the building block for all one-to-many and many-to-many communication patterns. Like multiplex views, multicast views are defined with a single, fixed component structure, and component views can be created dynamically. However, each component of a multiplex view represents a distinct medium passing through the view, while all components of a multicast view represent a single medium. Messages arriving at any multicast component on one side of an interface depart from all components on the other side. In Figure 2.6, a message sent by *Multi* will be delivered to both *Uni-1* and *Uni-2*.

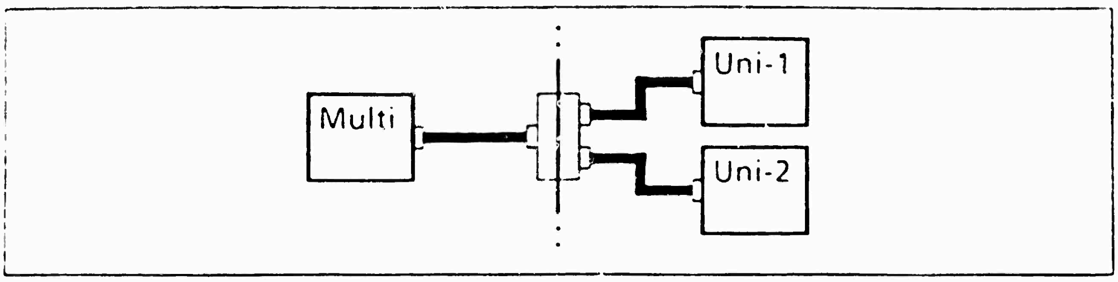

Figure 2.6. Multicasting

2.1.4. Non-Hierarchical Structure

So far, our structural representation allows only trees. Every space defines a purely functional composition of the adjacent subtrees. From the viewpoint of a functional programming purist, this purity of composition is very nice. From the viewpoint of a practical system designer, it can be more trouble than it is worth.

To use a module (object) in a purely functional fashion, its user must provide explicitly all needed external resources. Uninteresting plumbing accumulates higher in the object hierarchy for the sole purpose of connecting global services to modules at lower levels. The housekeeping and clutter hide the structure relevant to the object's behavior.

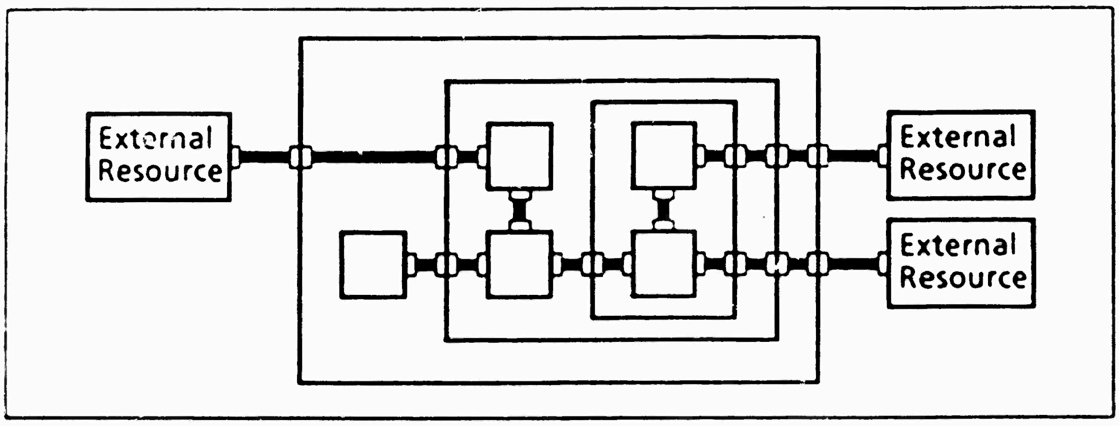

Figure 2.7. Clutter in a Strict Tree

The problem is worse in an open system. To make use of a newly installed service in a purely functional system, the shells between the service and its client must be ripped out and recreated with an additional interface to pass the new service. This is clearly inadequate and motivates violations of the strict hierarchy.

To provide the necessary escape, we let any two complex spaces be joined by a *splice*. For communication, a splice behaves just like a shell, providing a set of interfaces that communicate between the spaces. However, both ends of a splice appear to be domain boundaries pointing "down". This preserves the appearance of a strict directed tree even when a splice joins two spaces in the same domain (even one space to itself). Chapter 5 discusses non-hierarchical structure further.

2.2. Structural Operations

2.2.1. Examination

An agent must know the current process structure before it can make sensible decisions about how to manage it. The **inquire** operation provides the necessary access to the HPC system's structural database. It takes a structural element, such as a shell or an interface, and returns information about its properties and its neighborhood in the process structure. The process structure is unmodified by examination.

There are specialized inquiries for:

- The parent and children of a shell
- The interfaces of a shell
- The parent and children of a view
- The shell of an view
- The view (if any) connected to a view
- The root shell and controller of a domain
- The kind of structural element represented by an arbitrary name

The data returned from these inquiries is sufficient to write simple tree-search algorithms to traverse a domain exhaustively. Different database access mechanisms could have been provided. For example, a snapshot of the entire domain could be provided for off-line consideration, but such snapshots may be arbitrarily large and become out-of-date with the slightest change to the domain.

Because **inquire** is an operation on structure, the protection system makes domain boundaries opaque. Only agents of a domain can examine that domain's internal structure.

2.2.2. Connections

Communication paths between processes are incrementally created and destroyed by creating connections between objects. The **connect** operation takes two views as arguments and creates a connection between them. The views must belong to the same space, be distinct, compatible, and have no existing connections. **Disconnect** is the inverse operations. It takes two views that must be joined by a connection, and removes the connection.

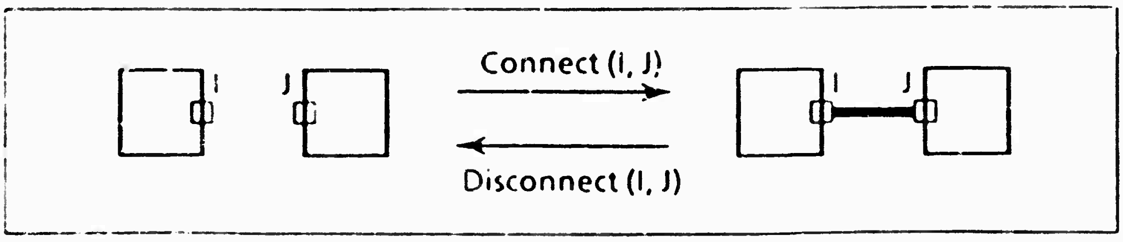

Figure 2.8. Connect and Disconnect

2.2.3. Media

When a series of **connect** operations has created a complete logical path between two processes, the HPC system creates a communication medium. The processes then communicate using the operations appropriate to the medium. Typical operations are **send** and **receive** for messages and datagrams, **write** and **read** for files, pipes, and streams, and call, accept and reply for remote procedure calls.

The HPC system makes these operations available for each communication mechanism it supports, but does not define their semantics or the behavior of the communication medium. Operations on media have no effect on HPC process structure.

2.2.4. Processes

To create a new process, the animate operation takes a empty object, a host identifier, a host-dependent image identifier, and a list of strings. The host is contacted, the image is started on the host, and the new process is placed in the empty object. The process receives the strings as initial arguments in a host-dependent way. There are two versions of the inverse operation. A process uses die to terminate itself. The kill operation terminates another process. Both operations take a simple domain as argument and destroy its internal process, leaving an empty object. (Processes are shown in light grey, empty spaces with a dot.)

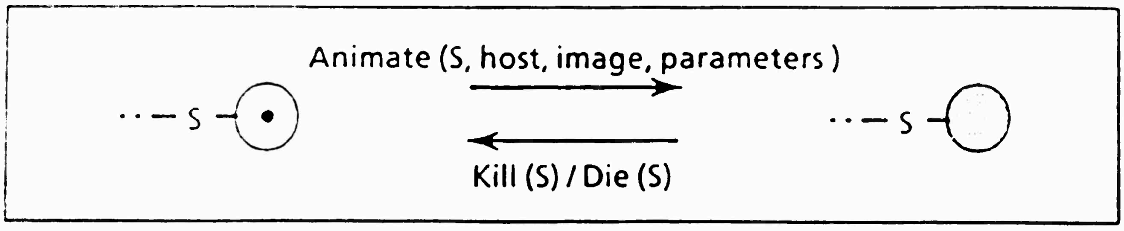

Figure 2.9. Animate, Die, and Kill

These operations also apply to complex objects. A process decides to become a simple or complex object during its animation in a negotiation with the HPC system. This decision is not visible to the agent invoking **animate**. To animate a complex domain, two new empty objects are created inside the previously empty leaf, a controller and the new process are placed inside the new objects, and a connection is created to join them.

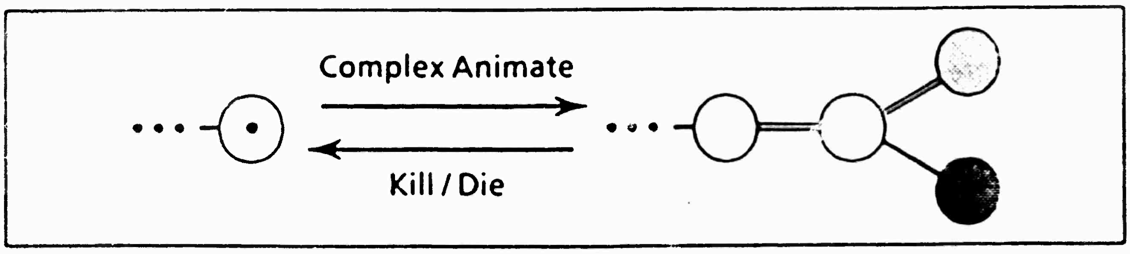

Figure 2.10. Complex Animation

As a convenience, kill and die can be applied to arbitrary subtrees. The entire subtree is removed, no matter how complex or how many subordinate domains are affected.

2.2.5. Shells

A shell is created explicitly by calling the enclose operation with a space, a partition of its incident interfaces, and a list of interface descriptions. There must be no connections between interfaces in different partitions. The space is divided into two spaces, both in the same domain, and one group of interfaces is moved to each space. The spaces are joined by a new shell with the desired interfaces. To destroy a shell using disclose, its interfaces must have no connections, and the spaces it joins must be in the same domain. Those spaces and their incident edges are merged together into a single space, and the shell is destroyed.

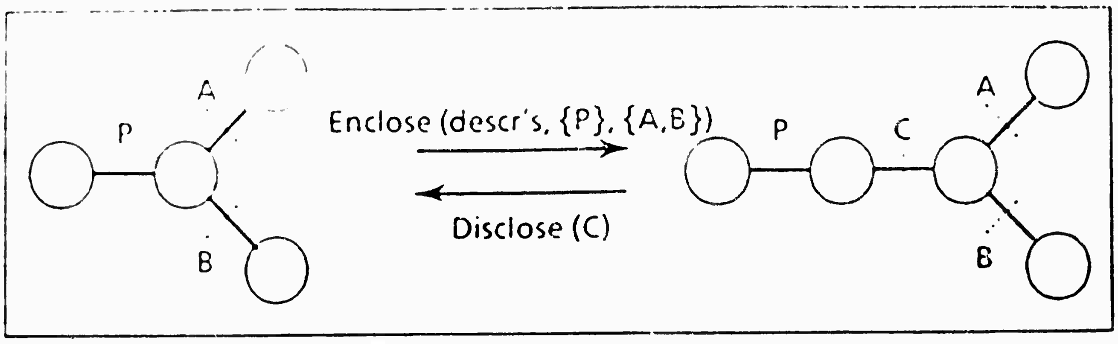

Figure 2.11. Enclose and Disclose

The names of these operations are taken from their effects in the hierarchical view. Creating a shell has the effect of surrounding the lower partition of shells, while destroying a shell releases the internal structure. And, because both a space and a bipartition are implicitly defined by a set of sibling objects in the hierarchy, the actual arguments for enclose are just a list of shells and a list of interface descriptions. Spaces are never explicitly manipulated in HPC, only shells.

2.2.6. Interfaces

Most interfaces are automatically created and destroyed when their parent or attached shell is manipulated, but multicast and multiplex interfaces have a dynamically varying number of component views. Each such component is created by the new operation on its parent view. A component view is destroyed by the delete operation. It must be a component of a multicast or multiplex view, and must not be connected.

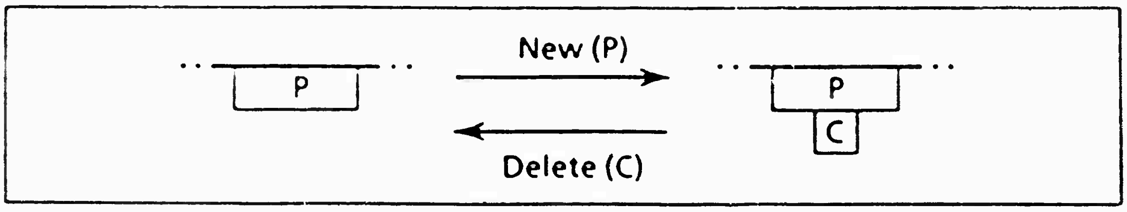

Figure 2.12. New and Delete

2.2.7. Domains

The domain creation operation, invest, takes a shell and an empty object as arguments. The object and the existing controller must be on opposite sides of the shell. The existing domain is reduced to the subtree on the controller's side of the shell, and the rest of the tree becomes a new domain. The controller for the new domain is placed in the empty object.

The abdicate operation takes as its argument an adjacent domain boundary to remove. The executing controller is removed, and the domain is merged with the adjacent domain. The depose is similar, but takes control *from* the adjacent domain instead of yielding control *to* it. Figure 2.13 illustrates these operations. The arguments for domain operations must be in the domain of the agent, just as for any other operations. An edge with an ellipsis (...) represents a sequence of one or more shells that must belong to a single domain.

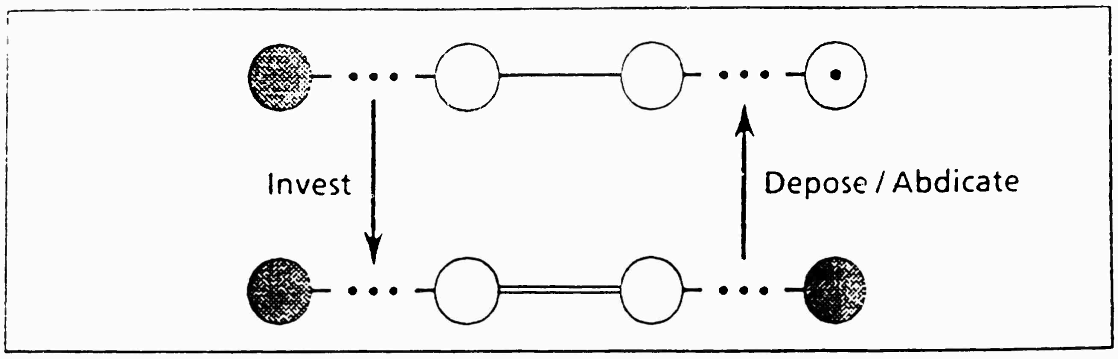

Figure 2.13. Invest, Abdicate, and Depose

2.2.8. Splices

Splices are created between existing empty objects using a two-step operation. Suppose objects *A* and *B* are to be spliced.

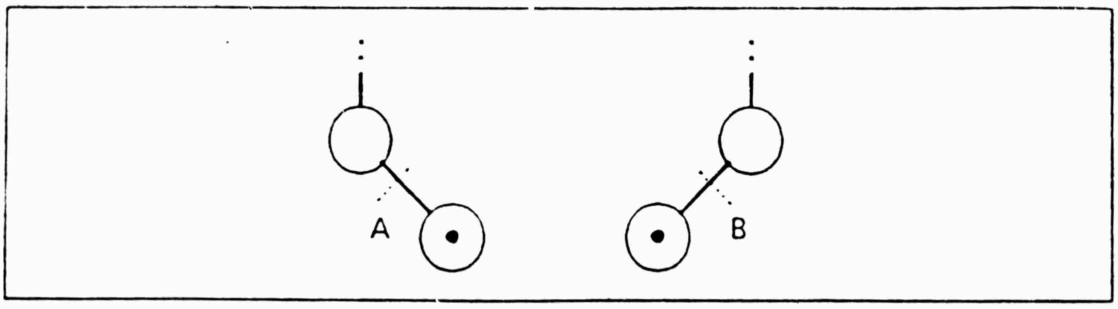

Figure 2.14. Splice (Before)

The agent for *A* invokes the splice operation with two empty shells. The existing shell *A* and space are replaced by the splice *C*, one end of which is hidden in a space accessible only to the HPC system. There is no immediate effect on *B*.

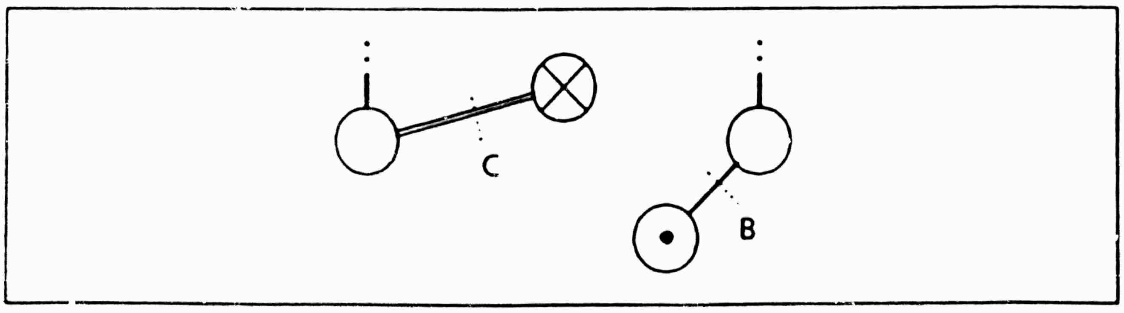

Figure 2.15. Splice (During)

When the agent for *B* decides to complete the splice, it invokes **splice** with the arguments reversed. The existing shell *B* and space are replaced by the previously hidden end of the splice *C*.

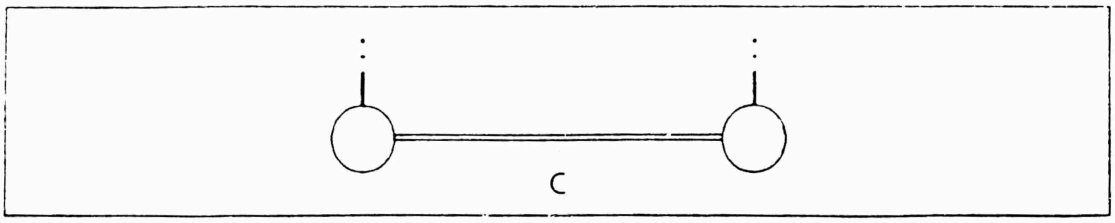

Figure 2.16. Splice (After)

2.3. Discussion

We begin to find interactions between multiprocess abstractions and the distributed environment already. Failures, concurrency, and dynamic structure lead to a model of interaction between application managers and the operating system quite different from most distributed systems.

A domain may have several agents (or a multiprocess agent) and the agent processes may choose to operate on a common piece of process structure. Executing on separate sites, their decisions are intrinsically asynchronous with respect to one another. In the absence of failures, cooperative action could be left to an explicit distributed consensus among agents, or provided in some form through communication between agents and the HPC system. However, failures of processes and hosts will occur at arbitrary times even if decision making among agents is synchronized. The environment is an additional agent with whom we can not debate or negotiate. As a result, an agent's view of process structure can become out-of-date at any time, even while trying to decide what to do about the previous state.

Transaction facilities are one conventional response to this possibility. They prevent the appearance of outside interference during a sequence of operations, but they do so by delaying or undoing operations in order to obtain the desired serialization. A transaction that is affected by a failure will be aborted, and the system reset to its original state. This is inappropriate for applications that must always make forward progress, are fundamentally asynchronous and therefore non-serial, must meet certain minimum performance requirements, or, most significantly, must *adapt* explicitly to failures and changing conditions. Obviously failures and other changes must not be hidden from applications in the latter class.

This raises a classic question: Should an agent monitor the process structure by polling, or should it receive an asynchronous notification of a change? Polling is obviously the wrong answer. An agent could easily spend all its time futilely looking for something that changed since the last time it was examined. Therefore, any system designed for adaptation to failure under user control must, at a minimum, provide asynchronous notification of the failure of processes and communication links (partition).

Any agent is a potentially distributed object, capable of carrying out several plans of action concurrently. If an agent blocked or was otherwise prevented from invoking an operation until some previous operation completed, this desireable concurrency would be eliminated. HPC obtains a more powerful system interface by allowing structural operations to proceed asynchronously with respect to the invoking agent, and using the asynchronous notification system to report the results of agent-requested operations as well as process and network failures.

The HPC system gives operations an at-most-once semantics within each partition, internally synchronizing when necessary to preserve the internal consistency of the process structure. An agent will see a serialization of operations, but it may be a different sequence for each agent due to asynchrony.

The disadvantage of this more flexible interface is that agents must be prepared for asynchronous interruptions of their plans and arbitrary delays while waiting for an operation to complete (or fail). Most distributed systems present a synchronous interface to their clients, usually in terms of local system calls on a host operating system, or remote procedure calls between application processes, or object-oriented operations on a distributed data structure. Asynchronous notification of changes in the environment (e.g., signals, upcalls, or callbacks) are usually restricted to special cases and circumstances. For example, under UNIX a process can receive notification that more data is available in a file, but not that the file has been deleted, renamed, or opened by other processes. This simplifies programming of the processes that are not involved in configuration and on-line management, but makes it difficult or impossible to write a manager process.

Chapter 3 Protection and Control Structure

3. Protection and Control Structure

The potential for change raises the questions of who may change a thing and what things they may change. Answers are usually given in terms of a standard protection model [Jon79], in which there is a collection of distinct *objects*, with operations that may *access* them in various ways. The *rights* to perform specific accesses on specific objects are collected into sets called *domains*. Privileges are distributed among active entities, known as *principals*, by associating a principal with a set of domains. A principal may carry out an access if one of its domains contains a corresponding right.

HPC does not require the full generality of the standard model. An object in the standard model is an HPC structural feature, such as a process, shell, or interface. HPC does not distinguish types of access, focusing on rights to specific operations, such as **connect(view1, view2)**. Domains are sets of rights, as in the standard model, and HPC agents are principals.

Policy and mechanism are clearly distinct in protection issues. A protection system provides a mechanism to enforce access control or information control policies established outside the system. These control policies are not arbitrary; they are based on the management of objects. Managers set policy and the implementation of policy requires action and change. Agents carrying out policy must be empowered to make the necessary changes. Conversely, an agent should not possess rights it does not need in order to implement policy.

Conventional protection systems (access control lists, capability lists) must allow for arbitrary collections of rights (and by implication objects) because there is no other mechanism capable of expressing the relationships among the objects in the system. This is unnecessarily general, because random collections of objects never have a common manager, while related objects often do. A contribution of HPC is the exploitation of the rich and explicit vertical process structure in the definition of protection domains. To date, protection mechanisms associated with hierarchically organized software have all been based on (static) scope rules for identifier visibility.

We believe that "controls", as a relationship, as important as "communicates with". It should be possible to build sophisticated control behavior out of less complex components, with the same compositional properties, and benefits, as communication behavior. HPC's second contribution in protection structure is the application of a powerful mechanism for identifying, composing, debugging, and controlling control behavior. These functions are rarely, if ever, available in conventional protection systems.

Section 3.1 motivates these contributions, building on the static domain, agent, and controller definitions presented in Chapter 2. We observe several interactions between the protection system, the hierarchy, and the requirements of distributed applications that can lead to internal contradictions and show how they have been avoided in HPC.

Section 3.2 continues the investigation of these interactions with the preservation of structural invariants by operations on structure. Creation and destruction of most structural features interact nicely with the protection system, but direct domain manipulation and process creation require special attention to avoid violating structural constraints and to limit the structural damage a malicious or erroneous agent can inflict.

User agent control of a domain may be lost, and some form of clean-up or recovery action must take place. Section 3.3 introduces the policies that the HPC system can be asked to apply automatically when user agents are unavailable, and shows how losses of control can occur either temporarily and permanently.

The basic HPC protection mechanism does not distinguish among types of access, or provide for arbitrary collections of rights. Because control is explicitly composed, user-defined objects can implement finer or more complex access policies, and transparently extend the system primitives, without additional support from the HPC system. (Section 3.4.)

The concluding Section looks briefly at some classic issues in protection, such as amplification and revocation of rights in the context of HPC.

3.1. Static Structure

The possible agents and rights of any HPC protection mechanism can be easily defined. Processes are the primitive active components in HPC, do all the work, and thus make all the control policy. By abstraction, an arbitrary object should be able to do anything a process can do, making objects the obvious candidates for agents. Applications are described in terms of shells, interfaces, connections, and processes. The protection mechanism is to control the rights to operate on these structural features.

A protection system also incorporates two relations, one between domains and rights, and the other between agents and domains. The invariant properties of these relations are the interesting features of a system. HPC tightly restricts the rights relation to follow vertical structure. Some necessary properties of the agents relation can be deduced from the principles of abstraction and composition, and from the need for redundancy and robustness.

3.1.1. Rights Relation

Arbitrarily contructed domains can and should be avoided. The grouping defined by vertical structure gives a strong guideline for the construction of domains. Grouping within a shell shows a coherent collective activity. If two activities don't interact directly, shells should be used to make their boundaries, and independence, manifest. When an application is structured properly, the contents of a space define a tight composition of cooperating objects that should be managed as a unit. Therefore, HPC defines domains at the granularity of spaces. The views incident on a space and the connections between them belong to the same domain.

Vertical structure also guides the clustering of spaces into domains. We do not expect common development and management of arbitrarily chosen pieces of an application. However, several levels of related abstractions that interact closely are often managed as a unit. For example, a program module may include many functions with independent interfaces, but the functions in a module are all related.

HPC exploits this locality in constraining domains to be disjoint, contiguous subtrees of the dual graph. Because domain contents are localized, domains are readily identified and traversed on the basis of local information. Every space knows which of its neighbors are in the same domain, and there is no need for an explicit list of a domain's spaces. Restricting a space to exactly one domain induces a coarse domain graph on top of the dual structure graph. This is an elegant relation that allows acceptably simple operations on domains and is sufficient for our target applications.

Other choices of domains are worth further study. For example, a nested relation on domains provides a protection model analogous to conventional programming language scope rules: inner agents (code) can affect enclosing structure (variables) while outer agents are more restricted. Operations on nested domains probably would be much more complex in order to preserve the more complicated structural invariants. This objection would not apply to the most general protection model with arbitrary domain overlaps, because there would be no constraints to preserve.

3.1.2. Agents Relation

The rights relation has remained stable throughout the HPC project, but an acceptable relation between agents and domains was more difficult to achieve. Two significantly different versions have been developed since the HPC protection system's introduction in [LeF85] and [LeF85].

There are several criteria to consider:

- Preserve abstraction

A complex domain should be opaque, indistinguishable from a process.

- Redundant control

Robust applications must have redundant agents to provide robust control, even if they have no other redundant components.

- Positive control

Loss of control should be prevented. Every domain should have at least one agent.

- Control structure

Control is a behavior as fundamental as communication, and should be subject to the same principles of manifest expression, abstraction, and composition.

- Control over control

Agents, domains, and the relation between them are not static. Changes to the protection relation, as well as process structure, must be protected.

- Simplicity

"Everything should be as simple as possible, but no simpler." -- A. Einstein. (Ironically, restrictions designed to simplify the protection system were at the root of many early problems.)

Primitive processes are opaque in HPC. In effect, every simple space (containing a process) comprises a domain, acting as its own agent. This interpretation is necessary if processes and (opaque) arbitrary objects are to be treated equivalently.

Because an agent may be a complex object, it is natural to obtain redundant control by propagating privileges to more than one of its components (via control composition). There may be several processes distributed across several sites, able to implement the agent's policy. As long as a single process remains, the domain is under positive control.

It is tempting to permit exactly one agent for each domain, using complex objects to obtain multiple agent processes. That restriction leads to a distinction between an agent object and the component processes that are authorized on its behalf. Propagation of this authority was a principal problem with one version of the protection system. Another reason to allow multiple agents for a domain is dynamic replacement of one agent with another. To maintain positive control, a new agent must be established before the old one is removed, requiring two agents at least momentarily.

We can also show that an agent must be allowed to control multiple domains to preserve the abstraction of robust applications. Consider an autonomous, complex object as an agent and assume that agents may control only one domain. (This assumption was the major problem with the other earlier version of the protection system.) Call the object, *O*, the domain it controls, *E*, and the domain of its internal structure, *I*.

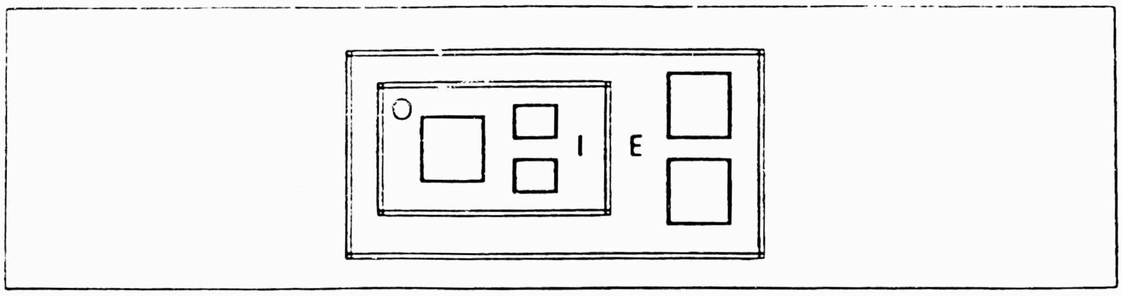

Figure 3.1. Robust, Opaque Agent

By assumption, the subobjects of *O* controlling *E* may not also control *I*. There must be a separate object to act as agent for *I*. Invoking the abstraction principle, autonomous agents should be allowed for *I*, since an individual process obviously must be allowed. If *O* is robust, its internal agent must be robust, therefore redundant, and thus an autonomous, complex object. This leads to infinite regress. If abstraction is to be applied uniformly, and robust control is to be possible, some agent must be allowed control of more than one domain. Specifically, it must be allowed control of itself, as well as an external domain. The interpretation of processes as their own agents leads to a similar argument: Some of the leaf processes of a complex agent must be agents for the domain it controls, as well as their own domains, because only processes can actually do work.

We investigated several rules for propagating the privileges of an agent object to its components, and concluded that a complex agent's control behavior should be defined using the same tools as a complex object's communication behavior. Propagation rules similar to programming language scope rules do not allow empowering only selected components. The hierarchical direction of propagation leads to unnecessary technical complexity and restrictions, and fails to preserve abstraction by distinguishing non-hierarchical structures from opaque subtrees. Positive control could be tested only by examining an entire subtree for live processes. Similarly, the full protection relation would not be manifest from looking at local structures.

3.1.3. Controllers

Composition of communication in HPC provides a clean implementation of a complex agent's behavior by selected components, manifest relationships, dynamically and incrementally defined paths, no hierarchical restriction, and the ability to debug at several levels of abstraction. The problems with hierarchical control propagation strongly suggest using the same tools to compose control. Agent processes are at one end of control paths and controllers were introduced to provide a explicit destination for such paths.

There is no way to disconnect and reconnect an interface atomically. For an agent to replace itself with another, the new agent must be connected to the controller before the old one is disconnected. This requires either a multiplex interface on the controller, or a multicast interface somewhere along the path, preferably on the controller so that direct connections can be replaced.

A special IPC mechanism named `control` is used as the structure of simple controller views, and controller objects present an interface with the following partial specification.

```
{ role: "HPC service",
  type: "multicast service",
  external: endpoint,
  structure: multicast
  { role: "HPC controller",
    type: "HPC invocations",
    external: extension,
    structure: control
  }
}
```

Control streams from the agents are merged into a single stream into the controller, and notifications from the controller are replicated for all the agents. Checking for positive control uses the mechanism for detection of end-to-end communication paths.

Agent privileges propagate across domain boundaries along `control` paths. There is no limit on the number of agents that may be connected to a controller, either directly or as multicast components inside a directly connected agent. Likewise, an object can have any number of `control` interfaces and can be agent to several domains. Activity on behalf of different domains is explicitly distinguished by using a separate interface for each stream of control messages.

3.2. Preserving Invariants

There are three ways protection relations can change. Agents associated with a domain can change, domain rights can change, and domains can be created and destroyed. Agents are defined by communication paths to controllers, and operations on connections or processes may affect the agent relation as side effects, but special attention is required only when the last agent for a domain is removed (to ensure positive control)

The direct effects of most HPC operations on the protection relations are minimal. Every view or other structural feature is always associated with one domain. When a feature is created or destroyed, rights to operate on it are added or removed from its domain. However, not all rights belong to a domain. Some types of access may require simultaneous access to multiple features, for example disclose requires access to both sides of a shell. A domain contains the right for a multiple access only when all the features accessed are in the domain.

Domain creation and destruction have more powerful effects. Only these operations affect the partitioning of features into domains. A feature is created in a well-defined domain, and it remains there until the domain is destroyed or a new domain is created around it. Rights can be added to a domain only by creating new features or by destroying another domain. However, the effects of abstract process creation and destruction have the greatest effect on all forms of structure, including domains.

There are three structural invariants that may be violated by kill/die and depose/abdicate. First, every process must be directly encapsulated within a domain boundary. Merging a simple domain with the neighboring domain above it violates this invariant by exposing a "raw" process. We cannot preclude this situation by restricting depose to complex domains, because this would violate abstraction. However, by interpreting a process as its own controller, it will be removed automatically when its domain is destroyed. Destruction of the superior domain is prohibited by an asymmetric constraint introduced below.

Second, every domain must have one controller. When **kill** is extended to arbitrary subtrees for convenience, it is possible to remove a controller, without removing its entire domain. This could be avoided by doing without the convenient extension. Alternatively, the operation could be restricted by an additional precondition. HPC preserves the invariant by destroying a domain as a side-effect of removing its controller. Any remaining contents are abdicated to a neighboring domain.

The HPC system distinguishes the root space and acts as agent for a domain consisting of just that space. If a top-level domain merges with the root, HPC kills the subtree and removes its shell to restore the desired root structure.

The naive definition of **kill** and **die** replaces the subtree on one side of a shell with a single empty space. The subtree that includes the root space also include all the top-level applications. Clearly, it should not be possible to destroy these from arbitrary places in the hierarchy. Similar considerations apply to **depose**. Repeated depositions would give an agent control over all process structure.

Hierarchical organization makes strong assumptions about the control privileges of superior and inferior levels. Components are usually considered implementing modules chosen by, and subordinate to, their parent. A superior level is expected to have the privileges to create and destroy inferiors, while inferiors are expected to have no control over superiors.

A single asymmetric constraint based on the hierarchy avoids interference between top-level applications and disruption of a superior domain by an inferior. An agent may **depose** or **kill** only inferior domains, and may **abdicate** or **die** only to the (unique) superior domain. Given this restriction, a single operation suffices for each of the **depose/abdicate** and **kill/die** pairs. These unified operations could be applied by agents on either side of a domain boundary to remove the inferior domain or subtree. (The HPC implementation uses such operations internally, but the pairs of distinct operations are retained in the application interface to increase the chances of detecting errors.)

3.3. Terminal Policy

An agent controls a domain when it has a logically complete and physically implemented communication path to the domain's controller. There are four ways to lose control of a domain.

- An agent causes the domain to be destroyed.

The domain's agents are forcibly taken out of control by the (possibly distant) side-effects of another agent.

- All paths from the controller terminate within the domain.

Only a connected agent can make a new connection inside the domain, and an agent can be added only by making one. Accordingly, if all the paths to the controller are broken inside the domain, control has been permanently lost.

- None of the paths from the controller into other domains is complete.

If some paths pass into other domains, but none are complete (there are no processes at their ends), control has been temporarily lost. Agents for other domains might complete a path and restore control.

- None of the complete paths from the controller are physically viable.

When no logically connected agent process is physically reachable due to a partition, control has been temporarily lost. When the partition ends, control may be restored.

The HPC system normally will do nothing without an agent. This is a (trivial) **null** policy. It is convenient to specify a more complex *terminal policy* to be applied directly by the HPC system during temporary or permanent losses of control. Such policies should be very simple.

Domain structure controls visibility as well as other access. Autonomous applications are opaque; their internal structure is completely hidden and complex applications can not be distinquished from single processes. This opacity is preserved by **die** and **kill**, because the internal structure is always removed before the domain is merged into its superior. However, **depose** removes control from the application and reveals its internal structure. Some applications may wish to hide their implementation under all circumstances. By specifying **die** as the terminal policy for permanent losses of control, an agent can ensure the privacy of its internal structure.

Policies stronger than **null**, but less drastic than **die**, must be applied to avoid subversion of access control, consistency control, and similar user policies during a temporary loss of control. For example, orphaned transactions should not be allowed unsupervised interactions with other processes. The **suspend** policy stops all communication passing through the domain. Even communication between processes unaffected by the physical partition is halted by forcing all interfaces across the domain boundary to the suspended state. (See Chapter 4.) Suspension lets component objects continue internal processing, but prevents any interactions between them.

Giving control to another user agent is also a suitable response. The system could **animate** a new agent object and connect it to the controller, or **abdicate**, merging the domain with its parent in the hierarchy. Trying to create a new agent introduces complexity. When this policy is selected, the HPC system must record the desired agent parameters *and* a fallback policy, as it may be impossible to create the necessary agent processes.

A complete terminal policy is a sequence of the basic policies **null**, **suspend**, **abdicate**, **die**, and **animate** (with parameters). The policy sequences for temporary and permanent losses of control are independently specified by a domain's agent. When a loss of control occurs, the HPC system applies each basic policy in turn until one succeeds. (The first four policies cannot fail.) A default policy is applied if the sequence is exhausted. The defaults are **abdicate** for permanent losses of control, and **suspend** for temporary losses.

The HPC system will not accept permanent responsibility for any domain, so **null** or **suspend** are not permitted in the terminal policies for permanent losses of control. However, any basic policy is permitted for temporary losses.

3.4. Policy Filters

Every control message requesting a legal operation that arrives at a controller is acted on. The HPC protection mechanism does not distinguish among agents, or among the legal operations, and therefore provides nothing like "read only" or "restricted" access to a domain. However, arbitrarily complex and sophisticated access control policies can be implemented without extending the basic mechanism.

Because command invocations are explicitly modelled as messages in a stream, they can be monitored and filtered. A trusted *policy filter* agent can be interposed between the controller and a *restricted agent* with limited privileges. (Figure 3.2.) It forwards authorized control messages from the partially trusted agent to the controller, and rejects unauthorized invocations. Notifications from the controller are forwarded in the opposite direction.

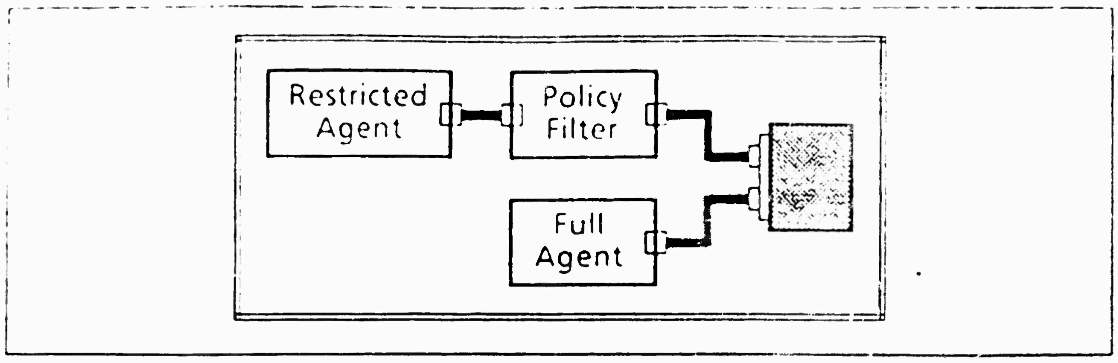

Figure 3.2. Policy Filter

A policy filter presents the same abstract control interface as a controller, although it does not have the same intrinsic robustness and availability properties. Controllers are placeholders for the HPC system and can be handled specially, while policy filters are ordinary objects and can fail independently of the HPC system. A reliable filter must be a complex object with internal replication, multiplexed delivery of incoming messages, and coordination between components to avoid multiple deliveries of the same operation to the controller.

Read-only access is easily provided by authorizing only **inquire** operations. More general forms of access control require checking the arguments of an invocation against a list of accessible structure. For example, it might be desirable to restrict an agent to managing the connections among a specific group of interfaces. The policy filter to enforce this restriction would forward those invocations whose arguments are on the specified list, and reject any others. The controller will reject illegal operations, so the policy filter doesn't need to make other explicit checks. Only notifications concerning the specified interfaces would be passed back to the restricted agent.

Policy filters need not tell the truth. For example, **inquire** can provide information about the parent and children of an interface, some of which might be inaccessible. A stricter version of the filter discussed above would tell the restricted agent only what it needs to know by modifying notifications to delete references to inaccessible interfaces.

A policy filter can erase all references to itself from notifications, report all connections to itself as connections directly to the controller, and translate all operations involving the controller into operations on itself. This technique provides a transparent illusion of unfiltered control, and permits user-level extensions to the HPC system. Most generally, a (hidden) policy filter can translate abstractions of new structures and operations into concrete HPC features the same way the HPC system translates abstract hierarchies into concrete host processes and media. The new abstractions need not have any strong relation to HPC at all.

A journal of invocations and their sources is one useful extension that requires no change to the HPC interface. A journal would allow transparent debugging and enforcement of audit trails. An extension that involves only a small change in the system interface would augment the host identifiers given to **animate** (*this host* and *specific host X*) with Medusa-style location specifiers like *same host as process P*, *very far from process P*, and *near process P but different host* [Ous81]. The hidden policy filter would evaluate the arguments of extended **animate** calls, perhaps with the help of a resource management utility, and then invoke the basic HPC **animate** operation with specific hosts. All other control messages would be forwarded without change.

Hidden policy filters are the natural interface between the HPC system and important administrative functions and access control features outside the system, such as native protection systems, charging, resource quotas, and classes of service. While the HPC system has no concept of authentication or user identity, and will simply report "process creation failed", an extended system could add additional properties such as (identity), new operations such as (login as user X), and extended reporting ("failed due to insufficient funds/privileges/resources").

The flexibility and power of policy filters is available only when the basic interface to system facilities can be intercepted and transparently replaced. Most system do not make this possible. The Accent operating system is one exception. An Accent *kernel port* is a system interface similar to the HPC controller, but interception must take place when the partially trusted process is created, or depend on that process to cooperate when reconfiguring.

3.5. Classical Protection Issues

Security policies are usually divided into access control policies and information control policies, and there are classical questions concerning the ability of a protection model to express and enforce them.

Because process state and the content of interprocess communication are outside the HPC system, HPC can not address the basic information control issues, which are *modification* and *spoofing* (how change to data is controlled), and *retention* and *confinement* (how propagation of data is controlled).

The primary access control issues are *propagation* and *conservation* (how privileges are transferred), *revocation* (how privileges are removed), and *mutual suspicion* and *amplification* (how two agents can grant each other only selective access). These problems have been examined most closely in protection systems that allow transfer of rights between domains (e.g., capability systems). The HPC domain system has quite different properties, because the rights in a domain are fixed, but access can be propagated and filtered without affecting the domains. The rights relation is modified in capability systems, while the agent relation is modified in HPC.

Amplification of rights is an important issue in capability systems where the owner of an object instance may not operate on it, while the manager for the object type does not have any rights for the instance. When the owner passes the object to its manager as a parameter, the manager is temporarily granted full rights to the object. Amplification has no analog in HPC because objects are never passed as parameters, rights are never transferred between domains, and there are no user-defined types.

HPC privileges at any moment are determined by the composition of agents and controllers. Connections and policy filters provide explicit propagation, mutual suspicion through filtering, and immediate revocation of access by disconnecting an agent. However, there is no way to restrict a partially trusted agent from propagating its current access further. This is consistent with the basic structuring principles. A strong conservation facility would violate abstraction by preventing a complex agent from implementing its control services any way it chooses.

Chapter 4 Communication Structure

4. Communication Structure

Connections and interfaces are the HPC structural features that tie processes and objects together into useful applications. Explicit communication is the only form of direct interaction between processes in HPC. Interfaces provide abstract destinations so that a process does not need to know the location or identity of its partners in communication, and the actual destinations are determined by chains of connections joined end-to-end at interfaces.

Because the possible patterns of interaction are expressed and limited by the available communication structures, HPC extends the intuitive one-to-one channels (simple interfaces) to multiple parallel channels (bundle and multiplex interfaces), and to many-to-many communication patterns (multicast interfaces). To illustrate the rich patterns that HPC can describe, we begin this Chapter an HPC specification for a replicated remote procedure call system taken from the literature.

Section 4.2 rigorously defines the significant communication paths, replacing the intuition of sequences of connections and incorporating the effects of complex interfaces. Section 4.3 presents the view properties intended for use by agents to manage structure in their domains.

There are several notable interactions between apparently independent features of communication structure, abstraction, and IPC implementations. For example, the information hiding provided by shell abstractions can be partially defeated by interface properties, and the hierarchical interpretation of nested objects is poor for labelling the direction of communication flow. Multicasting offers yet another set of problems. These issues are discussed in Section 4.4.

Section 4.5 concludes with a disection of communication into three functions that are quite distinct in HPC, and a comparison with the ISO reference model.

4.1. Example: Circus Replicated RPC

The Circus system extends the Courier RPC mechanism to groups of replicated processes [Coo84]. Circus will serve as a good demonstration of the expressive power of communication structure and hierarchical process composition in HPC.

In Circus terminology, a replicated group of processes is called a *troupe*. All members of a troupe are functionally equivalent. They may run at different speeds, and have different internal states, but must execute the same sequence of RPC operations. An RPC call from any member of a client troupe is replicated to every member of the server troupe. Similarly, a reply is replicated to all of the callers. Code in the Circus RPC library counts the number of requests or replies and applies various redundancy policies, like majority voting or quorum consensus, to decide if a valid replicated call has taken place. A special troupe, called the *ringmaster*, monitors the number of members in each troupe, which may vary dynamically, and makes this information available to the RPC library so it can check for the required degree of redundancy.

The Circus communication pattern can be expressed using a combination of multiplex and multicast interfaces, and troupes can be composed using a single connection. Consider Figure 4.1, which shows a single client connected to a multiplex server. Partial specifications for the client and server interfaces are:

```
[ role: "function stub",
  structure: simple Courier client
]


[ structure: multiplex
  [ role: "function entry",
    structure: simple Courier server
  ]
]
```

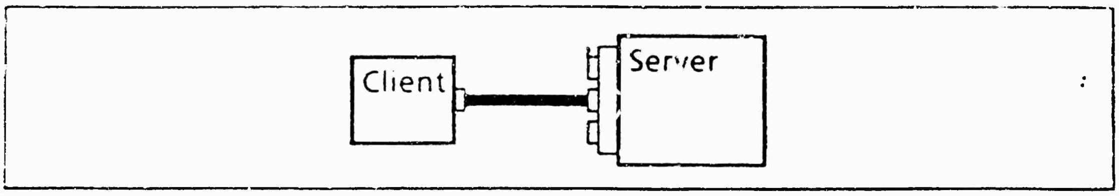

Figure 4.1. Simple Client and Server

Circus troupes are modeled as HPC objects, their member processes are simple objects within the troupe. The replication of RPC calls requires one minor change to the specification of individual troupe members. The RPC library expects sequences of calls rather than single calls, so we will use the type `replicated-Courier` rather than `Courier`. The relevant specification of client troupe member interfaces is thus:

```
[ role: "function stub",
  structure: simple replicated-Courier client
]
```

For server troupe members it is:

```
[ structure: multiplex
  [ role: "function entry",
    structure: simple replicated-Courier server
  ]
]
```

Each troupe replicates communication for each of its members, so the interfaces for each troupe should be multicast. The interface for the overall client troupe is specified as:

```
[ role: "troupe stub",
  structure: multicast
  [ role: "function stub",
    structure: simple replicated-Courier client
  ]
]
```

Each server component must be compatible with this client structure, and the server must multiplex its service to many clients, so the specification for the entire server troup interface is a multiplexed, multicast RPC interface.

```
[ structure: multiplex
  [ role: "troupe entry",
    structure: multicast
    [ role: "function entry",
      structure: simple replicated-Courier server
    ]
  ]
]
```

When a new client is added to the server, components are created on both sides of the server's main multiplex interface. (These components are multicast views.) The manager responsible for the internal structure of the server troupe creates one component of the internal multicast view for each individual server process. (These components are replicated RPC views.) It then takes each of the individual server processes in turn and creates a new replicated RPC component for its main multiplex interface. Connections are then created between the external RPC views of the server processes and the internal RPC views of the troupe.

A server troupe with two members and a client troupe with three members are illustrated in Figure 4.2. The troupes in this Figure have the same relationship as the individual objects in Figure 4.1. In both cases, a multiplexed server with components for three clients is shown servicing one. The only difference is that the client troupe interface is a multicast RPC interface, instead of a simple RPC interface, and the server is, of course, compatible. The individual client and server processes have the same interfaces as before, except for the expected replication of calls.

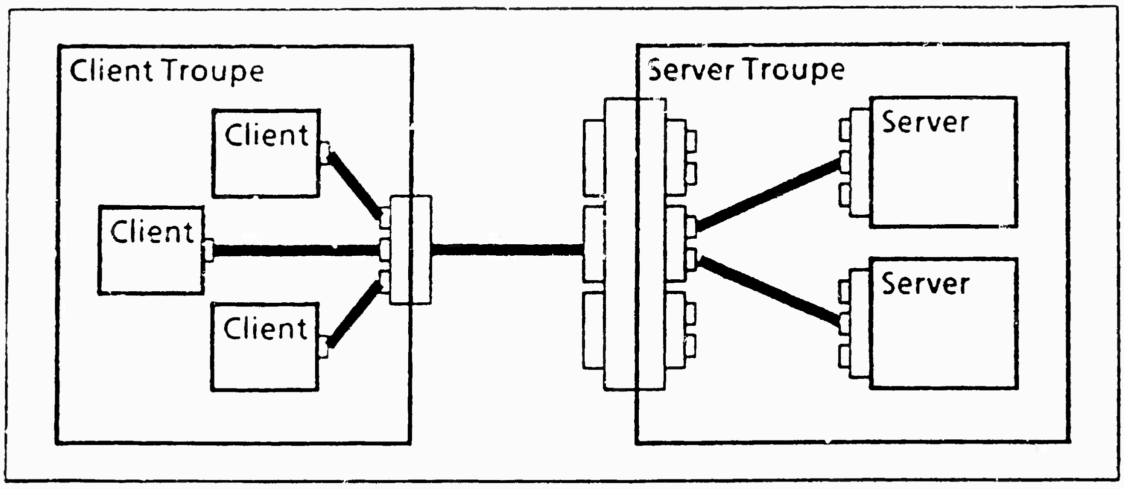

Figure 4.2. Circus Replicated Client and Server

Using HPC multiplex and multicast views, Circus troupes can be created using almost unmodified conventional clients and servers. Any degree of replication within a troupe is supported, and greater flexibility is possible than in the original Circus structure, because server troupes can chose to assign differing sets of server processes to service different clients, by controlling the server processes connected to a given internal multicast view.

Since the ringmaster is just a troupe, albeit a special one, the same communication structures could be used to communicate between troupe members and the ringmaster. A typical client/server relationship (especially for global services like the ringmaster) would be implemented using these multiplex and multicast interfaces on splices across the hierarchy, rather than through connections between clients and the server.

4.2. Significant Communication Paths

Earlier presentation of communication paths leaned heavily on intuition to simplify discussion. At the expense of some additional terminology, we present here the remaining details. The most important detail is the recursive definition of end-to-end chains, which are the only communication paths where action at one end can be reflected at the other. The concepts of endpoint/extension and of corresponding components are needed for this crucial definition.

4.2.1. Endpoints and Extensions

The views "at the ends" of a communication path, and those "in the middle" are distinct. The **send** operation can be sensibly applied to the former, while **connect** makes sense for the latter. We call them **endpoint** and **extension** views, respectively. Every view is created as one or the other, and it retains that property throughout its lifetime. The fixed distinction between endpoints and extensions improves abstraction, simplifies the system implementation, and eliminates a class of inconsistencies due to network partition.

Extensions are related to endpoints somewhat the same way opaque abstract data types are related to their concrete representations. Primitive behaviors are implemented at endpoints in terms of message contents, but the behaviors are composed and combined at extensions without access to the internal contents of the communication. Specifically, operations like **send** and **receive**, and the HPC primitives **new** and **delete**, can be invoked only on endpoints, while **connect** and **disconnect** can be invoked only on extensions.

Additionally, while both endpoints and extensions may be complex (with component structures), component views are elaborated only at endpoints. The hidden component structure of an endpoint is said to be *masked*. Consider the Circus server interface given earlier in more detail. (Figure 4.2.)

```
[ internal: endpoint,
  external: endpoint,
  structure: multiplex
  [ internal: endpoint,
    external: extension,
    structure: multicast
    [ internal: extension,
      external: masked,
      structure: simple replicated-Courier server
    ]
  ]
]
```

There are three levels to this interface tree structure. At the top level, the multiplex views on both sides of the shell are endpoints, and their components are visible. Each of these components is a multicast view, but the external views are extensions while the internal ones are endpoints. On the outside, this is the lowest accessible level of the view hierarchy because the third level structure is masked, while inside, the third layer is available as simple extensions of the multicast views.

Besides the abstraction benefits, allowing messages or procedure calls only at views with certain fixed properties eliminates the need to implement media for all communication paths. Only a subset of paths terminate in two simple endpoints, and only this subset requires the manipulation of physical media. In fact, we go further and implement media only for paths where both endpoints are internal views of real processes, and when both processes have expressed an interest in actually using their endpoints. This allows the HPC system to prepare the transport media used by a given process at convenient times.

It also means that a complex object can not send a message directly, enforcing passive hierarchies. (An agent could perfectly well **send** on any interface in its domain without this restriction.) All communication is performed by simple processes and the connections inside a complex object determine which subobjects, and ultimately processes, communicate on its behalf.

A chain would implicitly multicast to all points along it without the restriction of delivery to endpoints. The situation for sending is analogous. Since most paths are one-to-one, the distinction between endpoints and extensions avoids unwanted generality (and implementation complexity), requiring explicit introduction of multicasting when it is desired.

Finally, a class of possible inconsistency due to network partition is eliminated, and the reconciliation rules are simplified accordingly. Because the endpoint/extension property is fixed for a given view, there will never be a conflict between its use with abstract connections (extension) or with transport media (endpoints). All inconsistencies in paths can be reduced to a single type of illegal structure: multiple connections to a single view.

4.2.2. Implicit New and Corresponding Components

Paths involving complex interfaces represent several component paths, and it is important to keep track of which component views at one end are associated with which component views on the other. Bundles have a fixed number of components, and they are associated in the obvious way. Multicast interfaces have dynamically created components, but they all represent the same communication channel. However, each multiplex component represents a distinct channel and it is necessary to identify the other end of the channel.

It would be pointless for the **new** primitive to create a unique, one-ended channel. There would never be any way to communicate. Instead, under certain circumstances **new** creates views for both ends of the new channel. When two multiplex endpoints are joined by a complete path, a new component is created for both endpoints. (Because of multicasting, components may be created for more than one remote multiplex endpoint.) This is one of the two cases where an HPC primitive can have a significant non-local effect. (The other is splicing to a promiscuous service.) The **delete** primitive affects only its argument.

We can now define the useful notion of *corresponding* components. Correspondence for bundle views is determined by the fixed order of the component structures; the *n*th component of one bundle corresponds to the *n*th component of another. Given two compatible multicast views, all components of one correspond to all components of the other. Correspondence for multiplex views depends on the order in which the component views were created; a multicast component corresponds to just those components that were created by the same invocation of **new**.

4.2.3. Peers and Chains

Views that have been bound together as a link in a communication path are called *peers*. There are links through connections and links through shell boundaries. Views bound by a connection are called *public* peers, since the binding is manifest whenever the views are visible. Views bound through shell boundaries are called *private* peers, because the binding is not always known, even to the owners of the views, due to protection and visibility boundaries. A communication path, or *chain*, is a sequence of alternating private and public peers.

End-to-end chains, which terminate at endpoints, are the most important. When their structure is simple, they must be implemented with transport media. When their structure is multiplex, they allow implicit creation of components. We often call the terminal pair of endpoints end-to-end peers. In Figure 4.3 *A* and *D* are endpoints, and *B* and *C* are connected extensions. <*A*, *B*> and <*C*, *D*> are private peers, while <*B*, *C*> are public peers. The only end-to-end peers are <*A*, *D*> , which are neither private nor public peers.

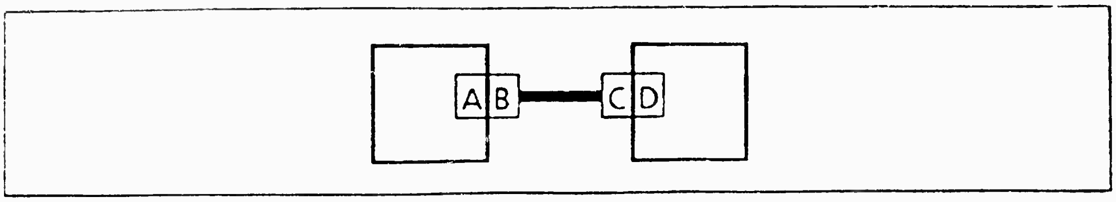

Figure 4.3. Simple Chain

Public peers are defined only by direct connections, but private peers are defined by a recurrence involving interfaces, chains, and corresponding components. In the base case, the two halves of an interface or complete splice are private peers. In the recursion step, corresponding components of end-to-end peers are private peers. This is another significant feature of end-to-end chains: private peers can cross an arbitrary number of shell boundaries. Figure 4.4 illustrates how these rules interact. Interfaces <*A*, *B*> , <*C*, *D*> , <*E*, *F*> , and <*G*, *H*> have relevant structure

```
[ internal: endpoint,
  external: extension,
  structure: foo
]
```

Interfaces <*M*, *N*> , and <*O*, *P*> have structure

```
[ internal: extension,
  external: endpoint,
  structure: bundle
  [ internal: masked,
    external: extension,
    structure: foo
  [ internal: masked,
    external: extension,
    structure: foo
  ]
]
```

By the base case, each interface defines a set of private peers. The public (connected) peers are <*B*, *I*> , <*D*, *K*> , <*J*, *E*> , <*L*, *G*> , and <*N*, *O*> . By definition, <*M*, *N*, *O*, *P*> is an end-to-end chain, and therefore <*I*, *J*> and <*K*, *L*> are private peers. From this, we obtain <*A*, *B*, *I*, *J*, *E*, *F*> and <*C*, *D*, *K*, *L*, *G*, *H*> as additional end-to-end chains. If foo is a simple structure, the last two chains are the only ones that might require implementation.

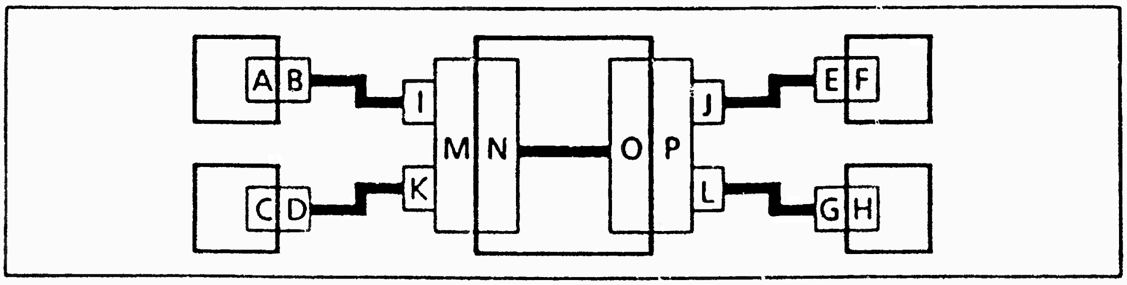

Figure 4.4. Complex Chain

4.2.4. Multiple End-to-End Peers and Chains

The semantics of one-to-one communication patterns are straightforward. An operation like **send**, for simple views, or **new**, for complex ones, invoked at an endpoint of an end-to-end chain has the appropriate effect at the other end. However, HPC multicasting allows both multiple end-to-end peers (Figure 4.5) and multiple end-to-end chains between a single pair of peers (Figure 4.6).

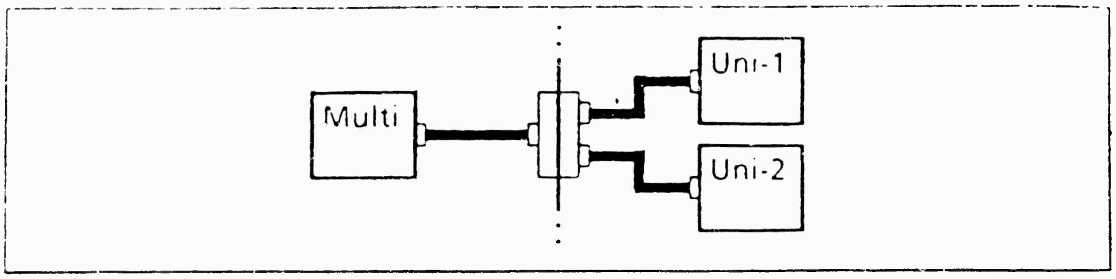

Figure 4.5. Multiple End-to-End Peers

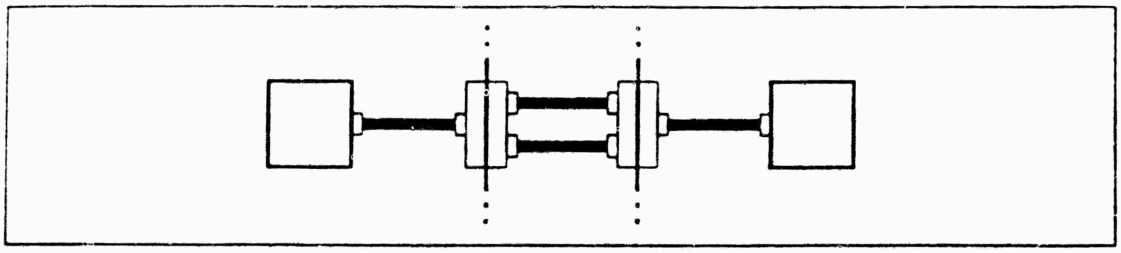

Figure 4.6. Multiple End-to-End Chains

This raises a very important question. Is communication associated with peers or chains? In HPC, chains define connectivity, not implementation, so one copy of a message should be delivered to each end-to-end peer, regardless of the number of paths to a peer (and subject to the semantics of the communication medium). Similarly for multiplex end-to-end chains, only one new component is created for each remote peer no matter how redundant the paths are.

The semantics of multiple peers and chains can be summarized as:

(1) Redundant chains between end-to-end peers are equivalent to a single chain.

(2) An operation at an endpoint is reflected, in a mechanism-dependent way, at each of its end-to-end peers.

(3) Operations at multiple end-to-end peers are reflected as multiple operations from a single peer.

4.3. Management Properties

Views have some additional properties that are maintained by the HPC system for the use of agents. These are static role and type labels, useful for identification, and a dynamic indication of reachability, useful for triggering application-level flow control and authentication. In general, however, an independent property service should make these properties available to clients, not the HPC system.

4.3.1. Role and Type

In addition to its structure (simple, bundle, etc), each view has a fixed role and type. Roles and types are arbitrary, uninterpreted strings of characters. Type describes the data representation or application-level protocol expected at an interface. Typical types would be `bytestream`, `rwho-update`, and `self-describing-datagram`. For complex interfaces, type generally describes the constraints or interactions between the component streams.

Role defines the abstract behavior presented or expected at an interface. The UNIX conventions of `stdin`, `stdout`, and `stderr` are well-known roles. The Accent `kernel port` is another standard role for a communication interface.

An increasingly common configuration is:

```
{ role: "terminal emulator",
  type: "X window client",
  structure: simple TCP/IP in-out
}
```

The complete description of the conventional UNIX interface to a process is:

```
{ role: "UNIX stdio",
  type: "stdio bytestreams",
  internal: endpoint,
  external: endpoint,
  structure: bundle
  { role: "stdin",
    type: "bytestream",
    internal: endpoint,
    external: extension,
    structure: simple UNIX-stream in
  }
  { role: "stdout",
    type: "bytestream",
    internal: endpoint,
    external: extension,
    structure: simple UNIX-stream out
  }

  { role: "stderr",
    type: "bytestream",
    internal: endpoint,
    external: extension,
    structure: simple UNIX-stream out
  }
}
```

Role and type properties provide a general mechanism for user-defined semantic interpretation of interfaces. Each agent is free to interpret labels as it sees fit and is not required to understand any particular label. Simple agents will check labels for conventional compatibility (e.g., role `stdin` connected to `stdout`). Sophisticated agents might interpose protocol or representation translation objects between views that are not immediately compatible (e.g., type `VAX-float-stream` connected through a conversion process to `IEEE-float-stream`).

A software development system that supports strong typing and separate compilation could generate distinct roles for each interface and use only agents that validate roles against a database before establishing connections between objects. Another good use for roles and separate interfaces would be to distinguish between the various entry points of an object accepting remote procedure calls or Ada-style rendezvous. A good development system can exploit the type information to make available at runtime a detailed description of the language and runtime dependent message types, remote procedure call arguments and return types, and so forth. This can be used by agents to help ensure the sensible interconnection of objects. Because interface type labels specify the proper interpretation of data transmitted through an interface, they are valuable in monitoring and debugging. Type information can make the difference between low-level packet traces and raw dumps and symbolic debugging of communicated data in a format relevant to the application.

4.3.2. Liveness

Usually an agent can not control or even trace a complete communication path, but even the most rudimentary control techniques require some indication that two processes are in contact with one another and remain so. A dynamic view property called *liveness* provides this indication based on *viable* endpoints, which are simple endpoints backed up by transport media and complex endpoints in complex domains.

While a view's structure is fixed, its liveness changes to reflect the communication paths that pass through the view. A viable endpoint is reachable through any alive view. When a viable endpoint is alive, a useful end-to-end chain exists. Flow control and (re)authentication procedures can be triggered by changes in liveness.

Liveness is computed by examining the chains that start with a view and continue with its private peers. The view is *alive* if at least one of these chains terminates in a viable endpoint, it is *dead* if none of these chains terminates in a viable endpoint, and it is *suspended* if neither of these two liveness values can be confirmed due to a network partition.

4.3.3. General Properties

There is no reason arbitrary properties couldn't be associated with interfaces, letting convention determine the significance of role and type. For example, the X window system incorporates a general facility for associating properties with windows. However, many things can have properties, not just HPC views or X windows. Property registration is a problem that should be solved once, not repeatedly, and that suggests a general context or property service, independent of the HPC agent or other services.

Role, type, and more general properties can be registered by clients without involving the HPC system. This applies to all static and many dynamic properties, most of which are uninterpreted by the HPC system. However, liveness is an example of a property with a non-trivial definition, interpreted by HPC, and with abstraction and protection boundaries that deny any single client access to all the data needed to compute it. Such properties should be computed by the HPC system, but made available to clients through the property service.

Expedient compromises develop in the absence of a property service. On one hand, a system that depends on a nonexistent service is not very useful, therefore the X developers provided the service themselves. On the other hand, a general property service is peripheral to the issues this dissertation is intended to address. Only *some* properties are needed to tell otherwise anonymous views apart. So, just two specific properties were built into HPC.

Properties are useful for objects as well as views. Consider UNIX filter processes, all with the same conventional interfaces and radically different behaviors. Role or a similar property would identify the function of a given filter. For simple objects (processes), several properties would be useful, including the physical location of the process, the image file from which it was animated, and its initial arguments.

The lack of object properties is especially acute during maintenance, as opposed to construction. As it stands, all processes are indistinguishable after animation. A post mortem examination of a failed process shows only an empty object. There is no record of the image that ran in the object, or what its parameters were, making it difficult to know how to repair the failure. We made no provision for recording properties of objects. This is arguably a significant oversight, but the *correct* solution is an independent property service not specific to HPC.

4.4. Discussion

Composition in HPC is expressed by its communication structures. The features of connections and interfaces have some unexpected interactions with each other, with the abstraction expressed by nested objects, with the protection system, with partitioning due to distribution, and with the semantics of real IPC implementations. We highlight the most important such interactions here.

4.4.1. Orientation

Until now, we have not been careful about specifying the orientation of simple structures. Surprisingly, there is no consistent way to label an object's interfaces such that both the description of flow relative to the hierarchical object, and the description of flow relative to the views, are intuitive and independent of context. This is a subtle issue that may lead to complex or error-prone programming. There are three related issues to clarify: the effect of the hierarchy on orientations, the flow direction specified by an orientation, and the side of an interface affected by an orientation.

If an orientation label is applied relative to an object, the labelling of Figure 4.7 results. These labels are appropriate in the hierarchical interpretation of process structure, and initially appear to be the right ones to use. However, Figure 4.7 shows that in is sometimes compatible with in and other times with out. Views can not be checked for complementary orientations without considering their relative positions in the hierarchy in addition to their labels. Worse, there is no local indication of the direction of communication flow. (Communication between an in and an out may flow in either direction, depending upon context.)

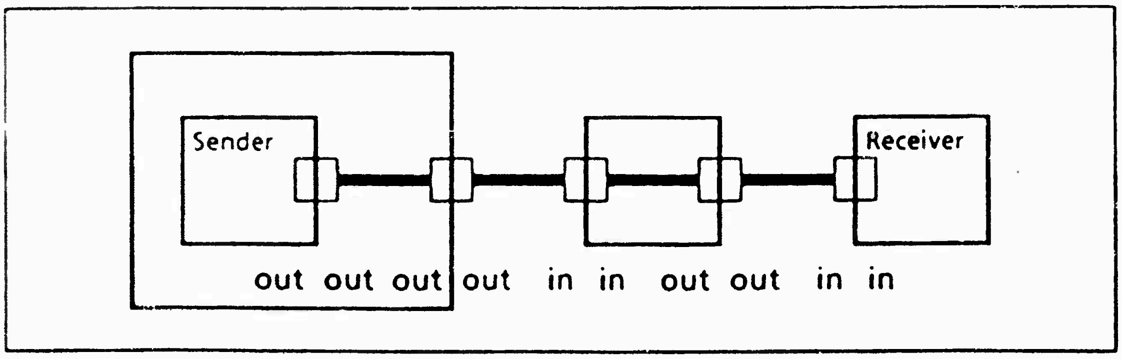

Figure 4.7. Orientation Relative to Objects

Automatic reconciliation of inconsistencies in the hierarchy due to network partitions sometimes requires taking a shell's parent and making it a sibling. This operation (Section 6.2.3) would be unnecessarily complicated by the need to reorient its interfaces to preserve complementary pairs of views.

If we apply orientation labels based only on the flow of communication and independent of the hierarchy, we have two more labellings as shown in Figures 4.8 and 4.9. The first labelling is appropriate for describing the flow relative to the external views of an object, while the second is appropriate for describing the flow from the internal point of view. It is clear that neither labelling intuitively describes the flow with respect to the other point of view. (In all three labellings, the actual flow of communication is the same.)

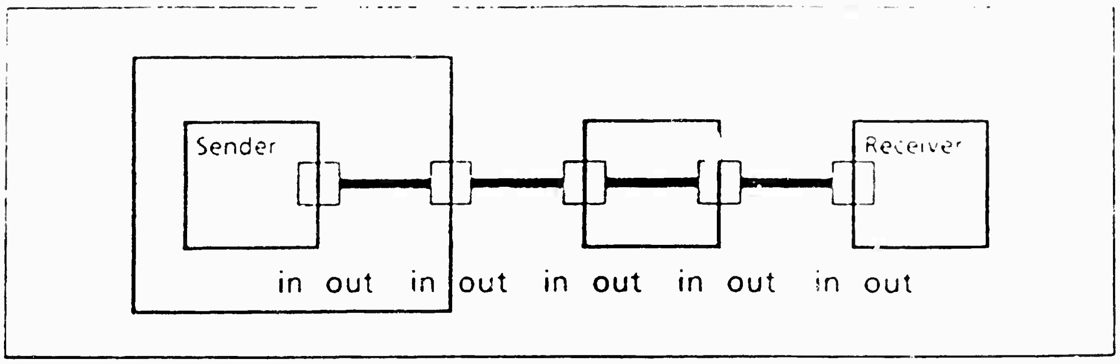

Figure 4.8. Orientation Relative to External Views

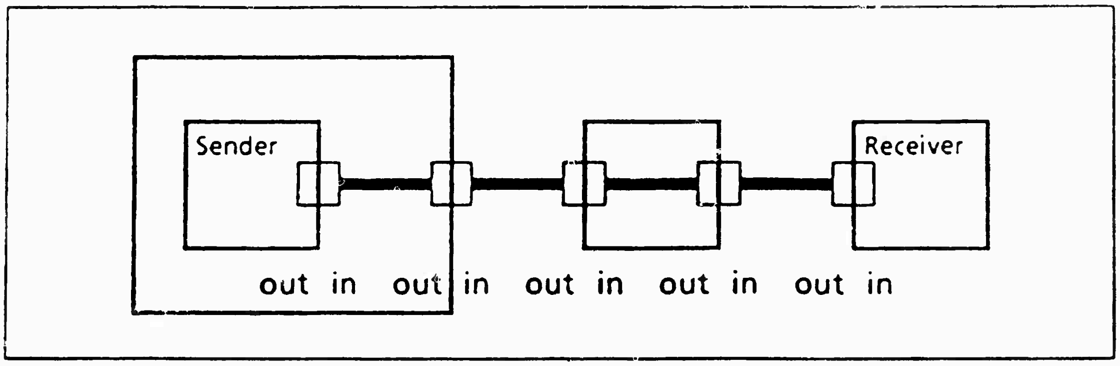

Figure 4.9. Orientation Relative to Internal Views

We select the labelling of Figure 4.8 because it provides the expected labels for the (external) abstractions presented by objects. The views of an interface (indeed, any pair of peers) have complementary orientations, therefore only the orientation of the external side of an interface must be given explicitly in an interface structure. Confusion over the orientations of internal views is possible, but there is a consistent rule governing communication flow. Messages flow *out of* of a view where receives are performed, and *into* views where sends are performed. For RPC mechanisms where the orientations are client and server, a client process makes calls on a server view, while the server process accepts and replies on a client view.

4.4.2. Endpoint/Extension Promotion

The endpoint/extension distinction has several attractive features, but it partially negates the information hiding provided by domain boundaries. We want to treat single processes and opaque complex objects indistinguishably. However, we also don't want to send on simple endpoints anywhere except inside a process, and we don't want to support connections and interfaces inside a process, at least not any further than its boundary to the outside world.

These are conflicting desires. If a complex object is animated inside a shell with a simple endpoint, it can't use that interface. If a simple object (process) is animated inside a shell with an extension of any structure, it can't use that interface. Conversely in either case, if the interface *is* used, the simplicity or complexity of the object can be determined from outside. The creator of an object should not know the complexity of its implementation or how it manages its internal communications, either before or after the object is created.

Our solution is to allow an object to promote its internal views to either endpoints or extensions, as desired. Promotion actually replaces the entire interface; when the object dies or is killed, the internal views do not revert to the original structure. The internal components of a complex view are created or destroyed (not masked) as necessary to match the change in structure.

To eliminate some inconsistencies that could arise during network partitions, the domain boundary shell and all its interfaces must be replaced whenever an interface is promoted. This preserves the invariance of a shell's interfaces, which is an important assumption of the conflict resolution procedures. To reduce the possible conflicts further, promotion is only allowed during the animation and investure operations. This is a simplifying feature, rather than a critical one.

4.4.3. Taps

Corresponding components are easily defined, but the *tap problem* illustrates a limitation with multiplex components. A valuable feature of dynamic communication structure is the ability to insert a monitoring process, or tap, at any point along a communication path to debug or filter the contents of the communication. Figure 4.10 illustrates the insertion and removal of a tap in the middle of a connection.

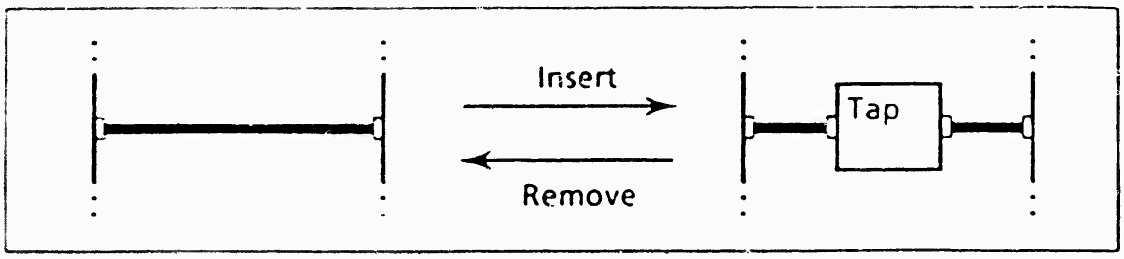

Figure 4.10. Tap on a Path

Ignoring changes in liveness, it is possible to insert and remove a tap transparently at any time on any structure *except* a multiplex interface. For multiplex structures, the tap must be inserted before a component path to be monitored is created and must remain in place until the last path has been destroyed.

For simple structures, taps forward communication from one side to the other. More generally, taps intercept the results of endpoint operations from one side and reinvoke the operations on the other. Of the complex structures, only *new* on a multiplex endpoint has an effect at its end-to-end peer. A multiplex tap detects the automatic creation of a new component on one side, and invokes *new* on the other side. This in turn creates a component at the ultimate end of the intercepted path. The tap remembers which components on each side match up as parts of an intercepted component path, so that communication on the components can be properly forwarded through the tap.

The problem comes in removing the tap. The obvious approach of disconnecting the tap and reestablishing the intercepted connection won't work because of the multiplex component correspondence rule. New views created while the tap is in place only have corresponding components in the *tap's* interface. When the tap is removed those views are permanently dead. Similarly, component paths established before the tap can not be monitored (except at the original corresponding components.)

We did not attempt a solution to this problem. Definition of corresponding component pairs by agents is a possibility worth investigation. Ideally, a solution to the tap problem would also address the cycle-avoiding and authentication problems discussed below.

4.4.4. Reflectors

Besides the multiple paths discussed earlier, multicasting can build some non-obvious communication patterns. *Reflectors* are one of these. Reflectors allow a chain to pass through the same view twice, once in each direction. (Figure 4.11.)

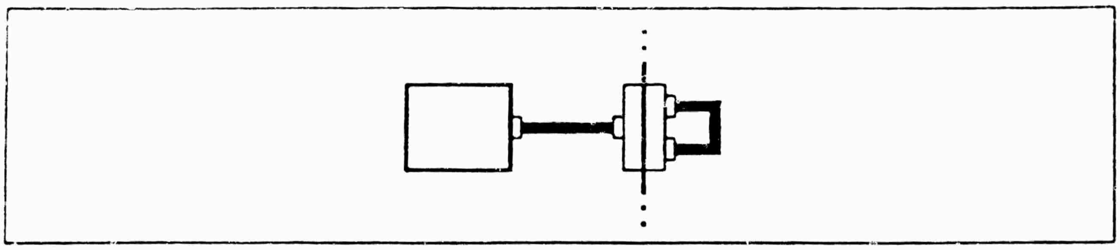

Figure 4.11. Reflecting Path

In order to connect the components of the multicast endpoint to each other, they (and all their component structures) must have a neutral orientation, such as *in-out*. This neutrality necessarily applies to the entire chain due to the compatibility restriction. Mechanisms lacking the ability to talk to themselves, like RPC, can not have neutral orientations.

Since reflection is just a specific consequence of multicasting, it has all the usual effects on component structures. Suppose an endpoint is one of its own end-to-end peers due to reflection. If it is a multiplex view, then a *new* operation on it will create *two* components. If it is multicast, communication through any of its components will be reflected through all of its components. If it is a bundle, the paths through each of its components will be individually reflected.

4.4.5. Cycles

A trivial cycle is shown in Figure 4.12. It has no endpoints and represents no path between processes. However, with multicasting, cycles can be introduced in the middle of end-to-end chains, as shown in Figure 4.13. For every such chain, there will be an infinite number of others, each with one more repetition of the cycle.

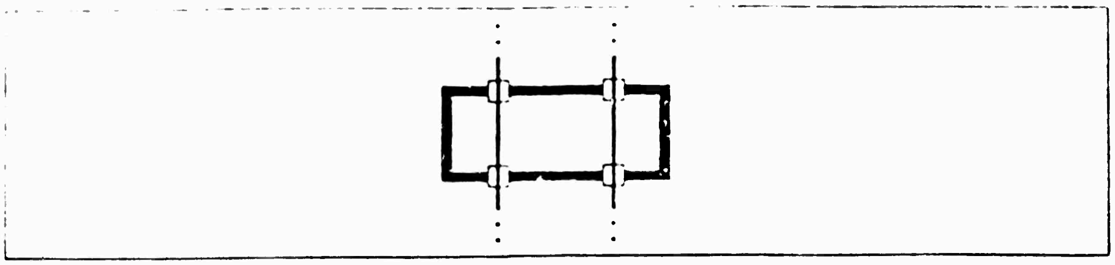
Figure 4.12. Trivial Cycle

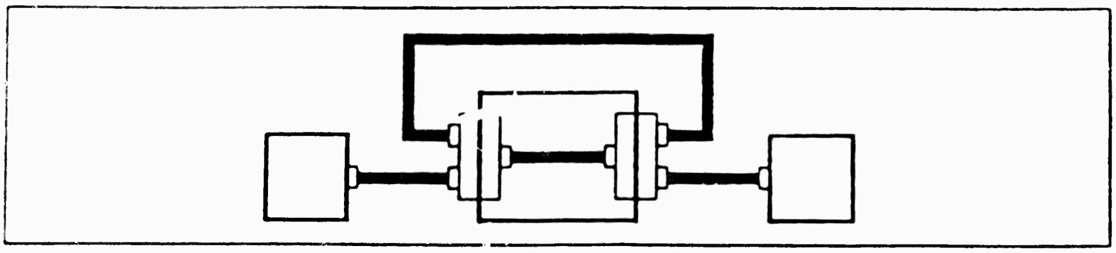
Figure 4.13. Complex Cycle due to Multicasting

These repetitions have no pathological effects on the HPC model, because redundant chains have the effect of a single chain. However, cycles, reflectors and multiple paths require a subtle algorithm to compute end-to-end chains without infinite looping or expensive checks for overlapping cycles and paths.

The prohibition of cycles might be suggested to simplify the system, because a system without them can express all the same useful communication patterns, but there is no obvious benefit to prohibiting cycles. First, it is the agent's job, not the system's, to determine which chains are sensible and which are foolish. From the system's perspective, the work required to detect cycles in order to forbid them is not less than the work needed to detect and ignore them. From the agent's perspective, liveness provides a rudimentary protection against waiting indefinitely for communication from useless channels. It must be admitted, however, that an agent requires more information to make a truly informed choice.

Forbidding cycles actually has some undesirable consequences. A cycle is detected only when the last link is created, typically due to a connect. Responsibility for the cycle (and the error) is distributed, but blame is not. There is no way to decide which agent should best take action, and the error is detected arbitrarily long after construction of the cycle begins. This is aggravated by network partitioning. During a partition the prohibition against cycles can not be guaranteed. When a cycle is detected upon network merger, it must be reported to an agent for removal. However, there is no obvious "last" link in the cycle and therefore no obvious agent to hold responsible. Yet removal of the cycle must be enforced (else cycles really *are* allowed). Most inconsistencies of this type are forced to a safe state by suspending all their views until an agent resolves the conflict, but that technique has no significance for cycles.

Forbidding cycles also means that operations like connecting two views are sometimes illegal based on information that is not available to the concerned agents. The cycle can involve masked structure that is not even accessible at the views in question. This problem could be avoided if agents had access to connectivity information that identified views at the end of (partial) chains in addition to the anonymous liveness property.

4.4.6. Authentication

Many access control policies are a function of a requester's identity as well as the type of access. There are two extreme views concerning the authentication of a communicating peer. The trusting view is that every connection has been made correctly. Every interface of a complex object is connected to a child that is authorized to receive or provide the corresponding service. As a result, processes are always put in contact with authorized peers. Authentication is a *de facto* property of process structure. Many hosts take this view regarding their machine console. Anyone at the console has privileges by definition.

The suspicious view is that an end-to-end peer could be anything at all. Through error or malice, every connection might be to an unauthorized peer and authentication must be carried out whenever a new peer is established. Most hosts take this view regarding their terminals. User identity must be established for each session.

Authentication procedures, encryption, and related topics are outside the scope of HPC, but we have a clear obligation to provide a mechanism to inform suspicious objects when a peer is (re)established and must be authenticated. Failing to end a session when a user loses telecommunication contact is a common security problem. The next user to establish contact with the host gains the privileges of the previous user.

View liveness is one mechanism for reporting changes in connected peers. For one-to-one chains, HPC guarantees that an endpoint will make a transition to the dead state whenever its end-to-end peer changes. Liveness can be too conservative to preserve abstraction, because a complex object can not change its internal implementation without triggering reauthentication. In principle, we should authenticate an abstract object, not the collection of leaf processes that happen to implement its services. If the object is trusted to provide the appropriate service, it should be trusted to manage its implementation.

In other cases, liveness is too weak for safety, because multicast interfaces allow the replacement of an authentication peer with an unauthenticated one without signalling a change in liveness, and because the addition of (unauthorized) peers after the first live peer is not reported.

In a trusting environment, the liveness property could be supplemented by notifying objects when their immediate external connections change. Neighboring objects could reauthenticate at the appropriate level of abstraction. (This mechanism would have to reflect changes in the degree of multicasting, as well.) In a suspicious system, one could record the partial chains reachable through each view. By associating an authenticated peer with a particular view on a particular chain (thus some specific object), substitution or insertion of an unauthenticated object on the near side of the selected view can be detected, while ignoring changes on the far side (inside the object).

Recording chains can be used to avoid cycles and address the tap problem, as well as to trigger authentication. Cycles can be detected or avoided by checking connected views for membership in each other's chains. If multiplex corresponding components are made explicit, some form of user selection of correspondence to address the tap problem becomes possible. Unfortunately, providing so much information about arbitrarily remote process structure to facilitate one aspect of security is difficult to reconcile with abstraction, information hiding, and access control. An adequate solution to these problems remains to be found.

4.4.7. Multicasting Semantics

HPC demonstrates that essential communication patterns can be expressed structurally. Multicasting is critical for many applications, and it has been easily integrated into the HPC model as a structural feature, rather than an addressing or special transport mechanism. Unfortunately, even simple uses of multicasting may not be compatible with the semantics of the underlying IPC mechanism.

```
{ structure: multicast
  { structure: simple TCP/IP in-out
  .
  .
}
```

Protocols such as TCP/IP designed for reliable data streams between two processes make very strong assumptions about exactly one set of peer processes. The HPC system can prohibit such mechanisms at the leaves of multicast trees, or do something more complex than create a single transport medium to implement the effect of multicasting. For TCP/IP, a central redistribution process is introduced to replicate and merge individual TCP streams. This provides a useful service, but it certainly doesn't provide the general semantics of reliable multiple delivery.

It is often not obvious how to use an existing IPC mechanism in an HPC multicast context. For example, the behavior of one mailbox shar d among all peers is not the same as a separate mailbox shared by each pair of peers. In the first case, only one peer removes a copy message, while in the second, all peers get a (separate) copy. There is not enough experience with multica nisms to select a general set of principles for extending conventional IPC mechanisms to multicasting. The interactions between multiple delivery and communication mechanisms with varying amounts of state will probably remain unclear in the foreseeable future.

4.5. Communication Taxonomy

Direct interactions between processes in HPC are determined by three factors: logical configuration, transport medium implementation, and communication. Each factor is controlled separately by distinct agents. Configuration is a dynamic, incremental process of modifying abstract structure and responsibility for one end-to-end chain is distributed over many agents. The HPC system is responsible for creating and destroying transport media to reflect the changing logical configuration. And the content of communication is controlled solely by the processes at the ends of the transport media.

In most systems these distinct functions can not be separated. Usually, configuration is merged with communication. For example, configuration in a link-based system (DEMOS, Charlotte, Accent/Mach) is accomplished by sending a link in a message, while connection setup in TCP/IP is controlled by the two communicating processes. A proper taxonomy of computer communication should distinguish these functions.

While the correspondence is not exact, in terms of the ISO seven-layer model, communication is user invocation of the transport layer, composition is user invocation of the session layer, and implementation is carried out by the session layer. (Table 4.1.)

HPC Feature	ISO Layer	How Related
role	application	purpose, intent
type	presentation	data encoding
composition	session	user invocation
implementation	session	layer function
structure	transport	user specification
signalling	transport	user invocation

Table 4.1. Relation to the ISO Model

Different IPC mechanisms generally have different operations for communication. For example, communicating with messages involves deciding when to send and receive messages, and what the contents of messages should be. Communicating with a semaphore involves deciding when to wait (*P*) and signal (*V*). Signalling with RPC involves deciding when to call a procedure, when to return from a call, and what the arguments and return values should be. (Table 4.2.)

Mechanism	Configuration Operations
HPC	connect, disconnect
CONIC	link, unlink
Hydra	connect, disconnect
RPC	bind
socket	bind, listen, accept, connect, disconnect
memory	link, load, address
file	create, open, close, inherit
mailbox	create, name
link	create, transfer
filter	set-filter
Linda	set-pattern

Table 4.2. Communication Operations

Mechanism	Communication Operations
HPC	*unspecified*
CONIC	send, receive, reply
Hydra	send, receive, reply
RPC	call, accept-reply
socket	*various, usually* send, receive
memory	read, write, P, V, fetch-and-phi
file	read, write, seek
mailbox	deposit, withdraw
link	send, receive
filter	send, receive
Linda	insert, remove, retrieve, eval-and-expand

Table 4.3. Configuration Operations

The configuration operations for these three example mechanisms are deciding whom to send a message to (whom to receive from), deciding which processes have access to the semaphore, and deciding which client stubs are bound to which server entries, respectively. (Table 4.3.) Implementation of conventional IPC mechanisms is usually triggered directly by configuration operations and managed by the host operating system.

Communication, composition, and implementation are not just conceptually different activities. If distributed programming is to incorporate more complex abstractions than the well-known client-server model, these activities must actually be carried out by different agents. We argue that configuration *must* be an activity distributed among multiple agents. Further, the configuring agents are generally *not* the same as the communicating processes. These statements are already true in simple ways for systems other than HPC. Designers of future IPC mechanisms should provide for their full realization.

To see that this functional separation is important to multiprocess, distributed systems more general than HPC, consider a system offering only one service with multiple server processes, a file service, say. Every process either belongs to, or is a client of, the file service, and the multiprocess implementation of the file service is transparent to the clients. Before a client and a server process communicate, the client must decide to access the file service and some agent in the file service must decide which server process is to handle the client's request. Neither the client, nor the file service agent, can decide unilaterally to bind the client and server processes. The logical path between the communicating processes has two segments. (There are two independent contributions to the decision to bind that particular pair of processes).

To extend this argument somewhat, assume that a protection system allows a process in one process group to access only public features (such as exported names or interfaces) of other groups, and that groups do not export the names of their internal processes. This already rules out the common situation (TCP/IP) where a communicating process *S* chooses its partner process *P*. If *S* and *P* are in different groups, *S* can only specify a public feature of the group G to which P belongs. The choice of P to complete the configuration must be made by some process in group G. This process can not be S and, symmetrically, P can not directly chose S.

Suppose the abstraction presented by a process group could be implemented in terms of other, less complex abstractions. This seems like a fundamental objective of any structuring system. One way to do this is allow nesting of groups, perhaps to bounded depth. Another way to achieve some of the same effect, even with a flat space of groups (no nesting), is to forward communication addressed to one group on to a second group, bypassing all the members of the first group. (A null modem has this kind of internal structure.) Access control or visibility constraints will prevent a client in one process group from knowing whether it is interacting with a process of a server group or some other group. In such cases, configuration can involve groups that do not contain *either* of the communicating processes. Establishing a path through several process groups requires the involvement, and implicit cooperation, of a configuring process in each of them.

Static process structures avoid run-time configuration choices altogether. Configuration is then typically an activity of human designers, while implementation is carried out by support software. Communications remains a run-time activity. However, static structures make the question of distributed and incremental configuration a moot issue of design methodology. We suggest that in successful methodologies configuration remains distributed and incremental, where distribution refers to separation among specification modules rather than process groups.

Chapter 5 Non-Hierarchical Structure

5. Non-Hierarchical Structure

Despite the familiarity of a strictly hierarchical structure, there are two reasons non-hierarchical relationship between objects must be accomodated. A strict hierarchy with explicit composition has nice formal properties, but is impractical for systems with real applications, even when their structure is static. Section 5.1 discusses the need for transparent access to shared resources.

Operations on HPC structure during a partition can easily lead to inconsistent hierarchies. Strict hierarchies are insufficient to express the merge of two strict hierarchies. A full discussion of this problem is deferred until the next Chapter, but the necessary tools are developed here.

The first step is separating shells' role in defining communication paths from their role in defining the hierarchy. Section 5.2 extends the dual graph of the object hierarchy with *splice* edges having no effect on the hierarchy, but allowing communication between arbitrarily distant objects. Splices are then carefully integrated into the protection system to preserve the local appearance of a strict hierarchy, and to prevent unwanted interference between distant objects. (Splices differ substantially from the mechanism originally described in [LeF85] and [LeF85].)

Managers of global services use a promiscuous splice facility to accept splices from arbitrary clients. This provides effective multiplexing of splices to complement multiplexing of interfaces. (Section 5.4.)

Section 5.5 concludes with some design interactions between splices and the existing structural features, and discussion of the pitfalls and tradeoffs of hidden violations of the hierarchy.

5.1. Transparency

The typical UNIX filter program has an input interface and an output interface, and only these interfaces are relevant to its composition with other programs. In addition, it may interact with resources like the file system. However, access to such resources is *transparent*[8] to the user of the filter. That is, access to external resources does not change how the user sees the filter. The filter provides a public abstraction (interfaces) and transparently makes use of additional private interfaces to external services. These implementation details are not part of the public filter abstraction.

A strict hierarchy with explicit composition has nice formal properties, but is impractical for systems with real applications. A purely functional methodology requires upper levels of the hierarchy to know, and provide, all the external resources required by lower levels. There is no way for an object to access a resource unbeknownst to its parent unless it complete encloses that resource. This methodology has some definite problems.

- Abstraction is unnecessarily limited.

[8] There is a regrettable clash in the use of the terms *transparent* and *opaque*. As ordinary words they are contradictory, but as technical terms they both indicate that certain details are not visible to the user. The operating systems community uses phrases like "virtual memory is transparent", while the programming languages community writes things like "this type is opaque." The recent interest in object-oriented systems has brought both communities, and their jargon, into intimate contact. We will consistently use "opaque" to indicate domain, and thus visibility, boundaries, and "transparent" to indicate that an object's abstraction, and thus its interfaces, does not define all of its interactions with external resources. If this muddies the water further, we ask to be forgiven.

A module's public interface should define its abstraction and hide its implementation. There is no way to separate the HPC interfaces used in "public" and "private" interactions.

- The benefits of explicit composition are lost.

At higher levels of the hierarchy there is an accumulation of uninteresting "plumbing" whose sole purpose is providing global services to lower levels. The connections relevant to the higher abstractions are obscured. (Figure 5.1.)

- The system is not modular.

Gaining access to a new service, or changing the implementation of an object, requires traumatic changes to the object hierarchy. All shells between a service and its client must be destroyed and recreated with a different set of interfaces to provide a different set of resources.

- The system is not open.

Top-level objects, including independent global services, are only trivially composed. No connections are ever made among them, and they can never interact.

These defects can be remedies by relaxing either strictly explicit composition or strictly nested abstractions. HPC provides transparent violations of the object hierarchy.

5.2. Splices

Shells normally represent boundaries or separations. There is an inside and an outside; the inside defines an object while the outside defines its environment. However, the dual graph emphasizes shells' role in communication. Communication between spaces must travel over the edges representing shells. We will now isolate this communication function, and unify the previously distinct structural features of shells and interfaces in the dual graph.

Each shell has a fixed number of attached interfaces, each of which is the root of a tree of views. A shell can be replaced by a pair of bundle endpoints bound as private peers. Its interfaces are demoted from roots to the

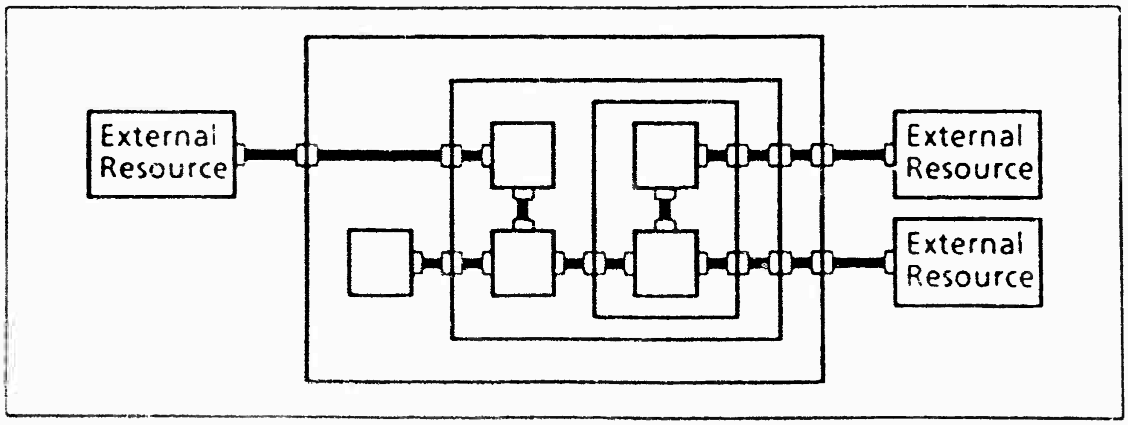

Figure 5.1. Unwanted Plumbing

immediate children of the new endpoints. Chapter 4's discussion of communication structure is unaffected. Besides the parsimony of features, this elimination of shells makes a nice sort of sense. A shell defines the abstract interface between two spaces by grouping together several communication interfaces, which is the function of bundles.

The HPC structure visible to clients retains the notion of shell, and the interfaces, rather than the bundle endpoints, are presented as the roots of view trees. However, internal computations of communication paths disregard shells, and use the bundle endpoints instead. The (internal) root of a view hierarchy is always attached to a single space, and all its descendant views are in that space. The root is always bound as a private peer to another view, and this binding defines a path of communication between spaces. (A related simplification is that there is exactly one direct private binding between view for each edge between spaces, rather than one for each interface on the shell. The corresponding component rule binds an number of interface subtrees.)

We can add now add arbitrary edges to the dual graph, even multiple edges between the same two spaces, in confidence that non-hierarchical abstraction and composition are well-defined. Each edge from a space represents one abstract interface between neighbors, and composition is defined by edges between spaces and connections within them An object can present one interface to its immediate parent in the hierarchy, others to its immediate children, and still others to unrelated objects through which it transparently accesses global services.

However, hierarchical organization should not be discarded simply because it can not be used everywhere. Violations of the hierarchy should retain the *appearance* of a directed tree by providing every edge with a direction and every space (save the root space) with exactly one parent. The second constraint means only a rooted spanning tree can be associated with the object hierarchy. (We continue to call its edges *shells*.) The remaining edges (*splices*) are fundamentally undirected, affecting communication paths but not the object hierarchy.

Splices are presented to clients as opaque subtrees to disguise the non-hierarchical structure. Both ends of a splice appear to be inferior domain boundaries, regardless of the real relationship between the endpoints. No violations of a strict tree will be apparent through inspection or traversal of a static object hierarchy.

These boundaries allow splices that join any pair of spaces, between domains, within one domain, or even a self-loop on one space. Splices within a domain must be accounted for because merging domains through **abdicate** or depose can easily transform splices between domains into splices within one. Similarly, merging spaces through disclose can transform splices between spaces to self-loops.

The apparent domain boundaries make it easy to provide dynamic behavior consistent with a strict hierarchy. The only operations applicable to an inferior domain boundary are **kill** and **depose**. The **kill** operation replaces an arbitrary subtree with an empty leaf space, and the HPC system is free to assume the convenient terminal policy **die** for an illusory domain, so **depose** is treated the same way. (Actually, creating and destroying a splice is a two-step process. Each endpoint is manipulated separately, but the entire splice is not destroyed until both endpoints have been removed from their incident spaces.)

Because creation of a splice must specify the remote end of the edge, it is a ripe opportunity for error and malice. A file system client can not be responsible for understanding the internal management of a global file service well enough to know where to put the other end of a splice. The agent for the file service should decide that. Nor is any agent allowed to unilaterally change the internal structure of unrelated domains that don't wish to be interfered with. Therefore, splicing is a two step process, requiring the active cooperation of agents controlling both affected domains

First consider the destruction of the splice <*A*, *B*> shown in Figure 5.2 by invoking **kill** or **depose** on the (apparent) shell *A*.

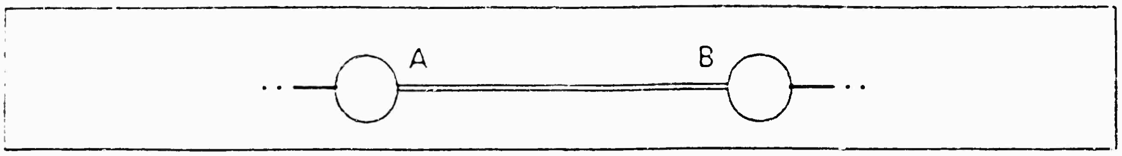

Figure 5.2. Complete Splice

The endpoint *A* is moved from its space to a hidden space available only to the HPC system, and replaced with the shell *C* and an empty leaf space (Figure 5.3).

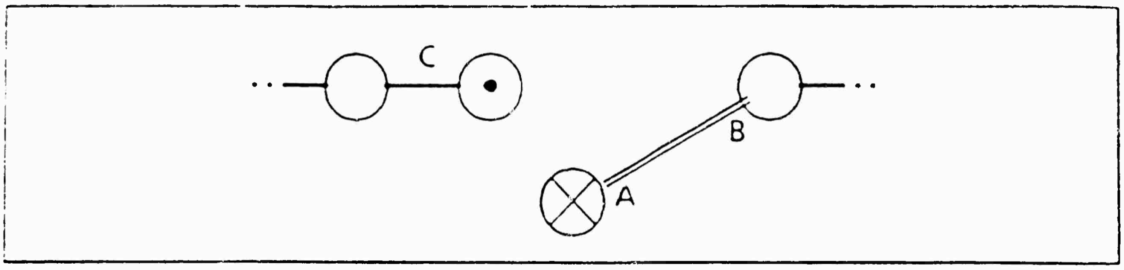

Figure 5.3. Incomplete Splice

When the other endpoint *B* is destroyed, it is similarly replaced by the leaf *D* (Figure 5.4). The splice is destroyed when both endpoints have been moved to the hidden space.

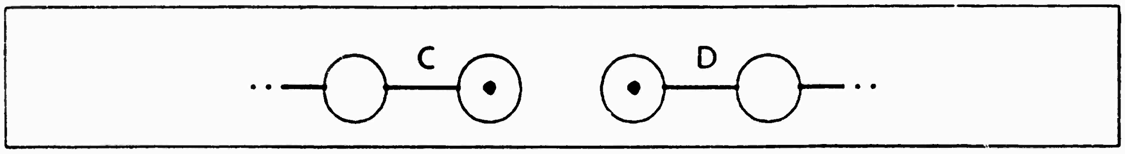

Figure 5.4. Unspliced Leaves

Creating a splice inverts these steps. To splice *D* and *C*, one agent invokes the splice operation with *D* and *C* as arguments (order is significant). *D* must be an empty leaf. It is replaced by the splice <*A*, *B*> with *A* hidden.

If *C* is an empty leaf with structure complementary to *D*, HPC remembers that *D* was to be spliced to *C*. When the other agent invokes **splice** with arguments *C* and *D*, HPC finds the hidden end of the splice <*A*, *B*> and replaces *C* with it.

If *C* and *D* are not compatible, the new splice will be created, but it will not be associated with the remote interface. (Cf. **new** on a dead or suspended interface.) This ensures structural compatibility for all private peer bindings, while preventing an agent from learning about another domain by blind probing for leaves.

The **splice** operation is cooperative, symmetric, and secure. The agents controlling both leaves must explicitly invoke **splice**, and it doesn't matter which invocation comes first. If one agent splices *D* to *C*, the *only* effect on *C* is that a subsequent **splice(C, D)** will complete the splice. If *C* is spliced to some other leaf, the incomplete splice from *D* to *C* is entirely ignored. The agent controlling *D* gains no information about *C* unless and until the agent controlling C chooses to complete the splice.

The splice and enclose operations are about equivalent in complexity. Both create a new pair of bound views and attach them to spaces. Enclose must partition a space, while splice must look for an incomplete complementary splice

5.3. Example: Accessing a Global File System

Accessing a top-level service from arbitrary points in the hierarchy is a primary motivation for splices. Here we use a file server application to illustrate how such access can be organized, glossing over the details of splice creation. In many file systems, a client must perform a directory operation, such as **open**, to obtain some abstraction or handle for a file through which file-specific operations, such as **read** or **write**, must be invoked. The natural HPC representation gives the client a splice to the file system directory, and an additional splice for each open file. The different types of operations available from a directory and from a file will be encoded in the interfaces of the splices.

The example client shown in Figure 5.5 will reduce a stream of data by adaptive filtering and issue a smoothed version of its output. Internally, it will use files to log filtering parameter changes, for off-line analysis of input characteristics and filter performance, and journal the input data currently within the filtering window, allowing the managing agent to recover or migrate the filtering process. Given an initial splice to the file system directory, the client can negotiate the cooperative creation of an additional splice by first telling the HPC system where its end of the splice is to be located and then invoking the file system **open** operation with its credentials and the name of the desired file.

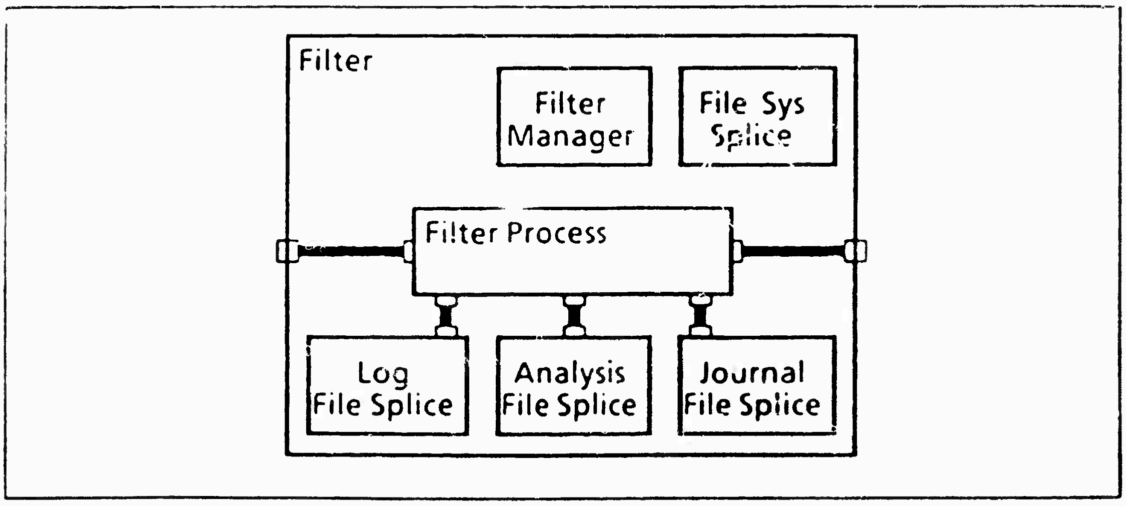

Figure 5.5. Hidden Access to Global File Service

If the file server finds the operation acceptable, it tells the HPC system to complete the splice. The server is free to do anything with its end of the splice. It may, for example, use a separate internal process to service each splice representing a file, perhaps distributed across hosts to provide the lowest cost communication with the respective clients. The resulting structure with separate server processes is shown in Figure 5.6. Note the client-server symmetry.

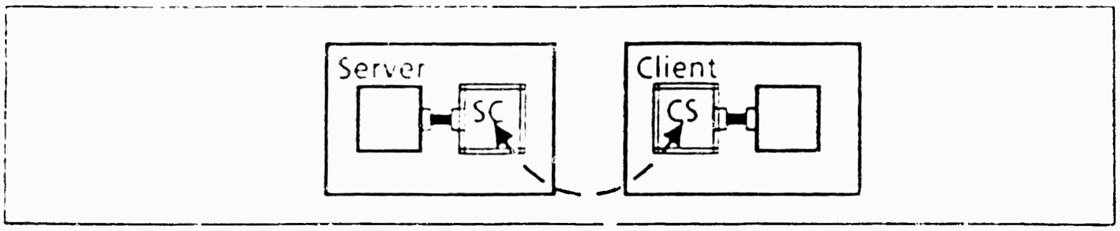

Figure 5.6. Client-Server Symmetry

5.4. Promiscuous Splices

The symmetric splice operation presupposes that agents can negotiate an agreement on the leaves to splice, which itself assumes some existing communication path. The "plumbing" needed for negotiating splices is not significantly less than the plumbing needed to access external services in the first place.

To provide an escape from this circular dependence, the HPC agent reserves a small set of *promiscuous* shell names for special treatment. Let *S* be a promiscuous shell, and *A* be a compatible leaf. If the agent controlling *A* invokes **splice**(A, S), the other end of the splice will be immediately installed as a sibling of *S*. With this special treatment, the agent controlling *S* does not need to take any action to create the splice.

(This operation, and **new**, are the only HPC primitives that can have a direct, unilateral effect on remote structure. Their ability to interfere with the remote domain is limited to the nuisance level because new structure is always created, and no existing structure is modified. Creating a splice or a new component therefore can not interfere with any ongoing activities involving other structure.)

These reserved shell names are the equivalent of well-known service numbers or well-known port numbers. In fact, this mechanism is taken more or less directly from the DARPA TCP/IP protocol for establishing a connection. A TCP/IP server listens on a socket with a well-known port number. Clients initiate connections between a local socket and the server's well-known socket. The TCP/IP protocol creates a new server socket and establishes the actual connection between the client socket and the new server socket, leaving the well-known server socket free for initiating other connections.

It is unnecessary, and ultimately painful, for every server to use a well-known name to advertise its services.[9] Here we demonstrate a a global name registry or switchboard service. Any object can create a splice to the switchboard using its well-known special shell and then send a message to register a component shell to be used for splices, or to request the switchboard to facilitate a splice with another object.

In Figures 5.7 through 5.10, a client is shown locating a server and creating a splice to it. Initially, in Figure 5.7, *Server* has spliced *SX* to the special switchboard shell *X*. (Dashed curves will indicate pairs of shells that have been spliced.) After creating the path to the switchboard, *Server* registered (by string or any convenient identifier) *SC* as a shell it is willing to splice to a client.

[9] The DARPA Internet experiences with this problem are reflected in the features of the new Internet "domain name server", which, for the first time, eliminates a global dependence on well-known, statically allocated, port numbers and host numbers.

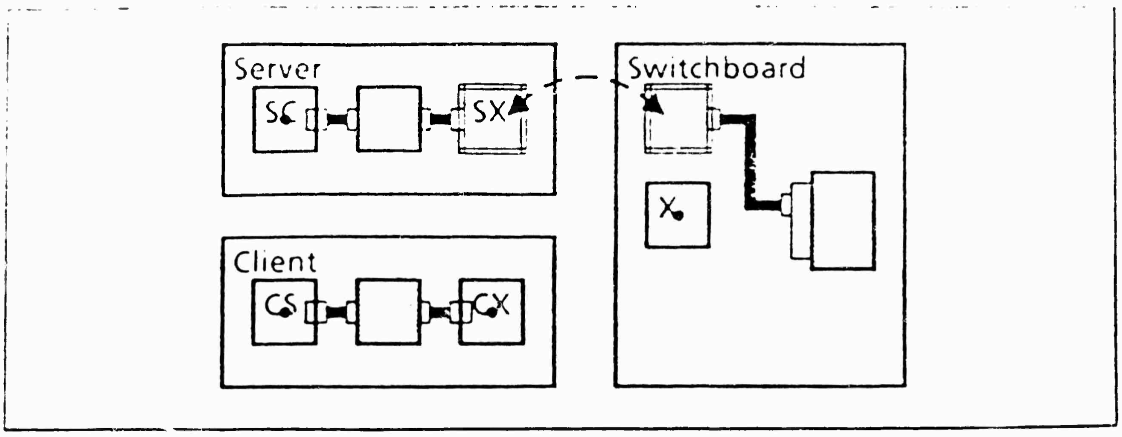

Figure 5.7. Server Registered with Switchboard

In Figure 5.8, *Client* has spliced its shell *CX* to special shell *X*. Notice that a new sibling of *X* is created for the splice. *Client* can detect when the switchboard connects a process to the new shell, as in Figure 5.9, due to the liveness property. At that time, it will register its shell *CS* as the shell it wishes to splice to a server.

Any time thereafter, *Client* can send another message to the switchboard asking for assistance in negotiating a splice to *Server*, for which *Client* knows the appropriate identifier. The switchboard response is to send *Server* the name of shell *CS* through the spliced shell *SX*, and to send *Client* the name of shell *SC* through the spliced shell *CX*. The two objects can then carry out the two step splice operation, as illustrated in Figures 5.2 through 5.4. The final structure is shown in Figure 5.10.

A point should be stressed here. The HPC agent supports promiscuous shell names, and permits the switchboard to use one. Other than that, all that has been described here is defined as an *application*. The interface to the switchboard, the messages to be used, the kinds of strings which can be used to register a shell, and so forth

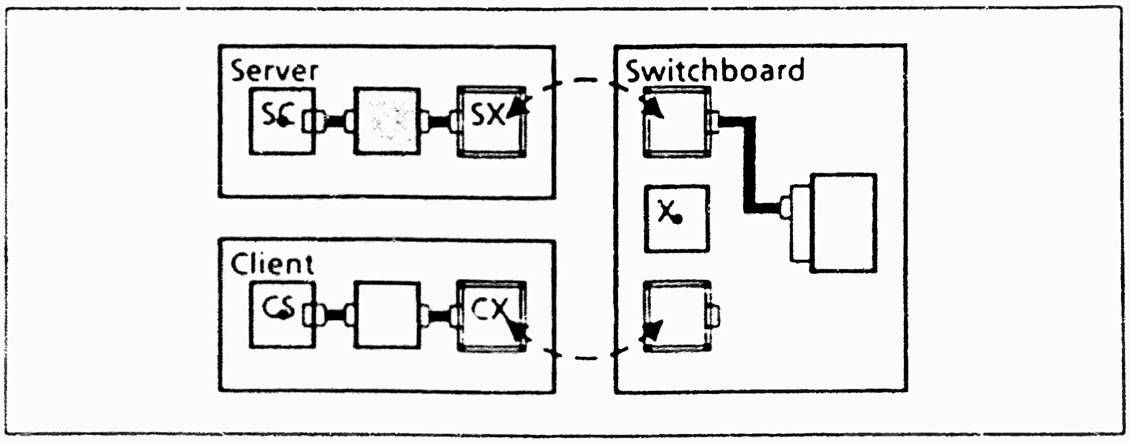

Figure 5.8. Client Contacts Switchboard

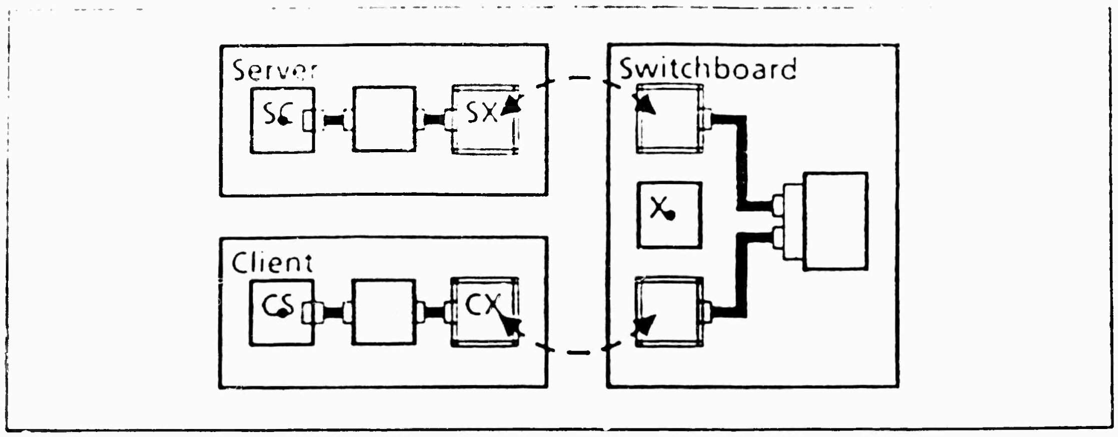

Figure 5.9. Switchboard Communicates with Client

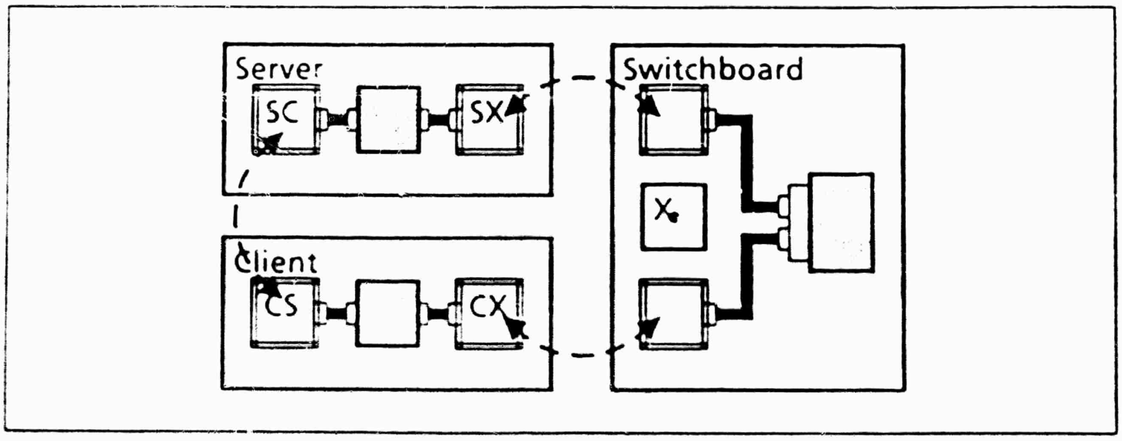

Figure 5.10. Client-Server Splice Established

have no impact at all on HPC. They are established by convention, and can be freely extended and changed.

5.5. Discussion

5.5.1. Transparency is Not Always a Good Thing

Transparent abstraction fulfills a pragmatic need at the expense of the aesthetics of pure composition. However, even from the purely practical point of view, hidden violations of the hierarchy can have disadvantages. Ignore software for a moment and consider apartment layout. Residents want hot and cold fresh water, sewage disposal, heating, and electricity provided without worrying about where these services come from. For its occupants, an apartment is a convenient self-contained unit independent of other apartments. However, for architects, contractors, and repair persons the bounds of a given apartment are somewhat artificial. They are much more interested in the network of plumbing and wiring that ties a whole building together.

HPC does not reconcile these two points of view. It allows construction of complex objects using either explicit provision of service from above, or hidden direct access to services. However, architects and building managers can prevent residents from tapping electrical trunks directly, and are obliged to provide residents with the standard services. In hierarchical process composition, an object can not prohibit a subobject from obtaining transparent access and a subobject can not force its parent to provide a needed service.

Future research should investigate the problem of controlling hidden access paths Residents of an apartment complex need to be prevented from tapping services directly for two primary reasons. First, resource utilization must be controlled and accounted for. The second issue is safety; Uncoordinated access to a resource can endanger all users of the resource. In large-scale software, similar issues arise. An object may need to restrict or account for the resources it consumes, and can only do so by controlling the access of its subobjects. Cycles, storage, and communication cost real money, and most hosts shared among several user groups have some form of accounting, billing and quotas which must be observed.

As an example of the safety problem, consider the lightweight transaction mechanism proposed by Zwaenepoel and Almes, which could be built easily on top of the HPC agent [ZwA85]. In their scheme, each worker process participating in a distributed computation is given a unique identifier, a set of input files, and a set of output files. A centralized job manager is responsible for assigning resources to workers and collecting their results. The manager gives a worker process unique temporary files to use for output, so a worker has no effect on shared data prior to a commit. When the manager decides to commit a computation, it (atomically) renames temporary output file as shared data files. Since a worker's results are completely written into a temporary file before a commit, the actions of a worker process appear atomic. If the manager process is aborted, the rename operation will never be executed, hence the workers will have no effect on shared data. If a worker process is partitioned from the manager, a new worker process can be created by the manager with a new identifier. The output files of the previous worker process will never be renamed since only the most recent identifier is allowed to cause a commit. Eventually the orphan worker process will complete or abort; In either case its results will be discarded.

As long as worker processes use only resources obtained from the manager, this form of lightweight transaction works nicely. However, if worker processes can transparently bypass the manager and obtain file access directly from the file system, the apparent atomicity of a transaction can not be guaranteed.

It may be possible to exploit the hierarchy when establishing limits on shells that may be spliced together. The following sketch attracts further consideration. An object could limit the subtrees (anywhere in the tree) to which its component objects may create splices. That is, a server can usually be identified with an object, and therefore a subtree of the structure graph. An object might allow its components transparent access to specified subtrees (services), or forbid access to certain services, and so forth. Existing experience with scope rules for programming languages with import and export should be quite relevant.

5.5.2. Peer-to-Peer Symmetry

Splices avoid the usual asymmetry between clients and servers. Returning to the principle motivation for transparent non-local access and thus the splicing mechanism, a client of a global service can splice a leaf shell to one belonging to the server and then communicate directly with the server. The client's spliced shell retains its structural position as an implementing component within the client. Non-local access through a splice appears to be access to a completely enclosed subcomponent.

It is a short step to realize that the server can think of a client as a subcomponent as readily as the other way around. This is an extension of the parent-child symmetry made apparent by the dual graph, and it holds even in the hierarchical view. Splices provide naturally for peer-peer relationships as well as the more common, but limited, master-slave organization.

The dual graph displays even greater symmetry. An object is simply a subtree and the abstraction it implements is the shell at its root. The global process structure can be broken into subtrees at any space and the composition inside the space defines the global behavior as a function of the subtree behaviors. Because there is no distinction between parent and child in the dual graph, a subtree can treat an incident shell (representing its parent) as a logically subordinate object, but the parent can treat the other side of the shell (representing the child) the same way. It is all a matter of perspective; the root of the dual graph can be chosen arbitrarily.

5.5.3. More and Fewer Restrictions

The strict tree structure could be retained by simulating the splice operation outside the HPC system. Specifically, instead of splicing two shells, an agent could animate an appropriate process in an empty shell, passing parameters to indicate the shells to be spliced. Processes of this type would interact outside the HPC system, locating processes with complementary arguments, and forwarding communication from one process to the other. (Splices between more than two endpoints could be simulated.)

As with multicasting, we are obligated to demonstrate how important relationships can be directed expressed in terms of process structure. Therefore, we integrated splices into the object hierarchy to express transparent, non-hierarchical access as directly as is consistent with protection and visibility restrictions, rather than depend on a mechanism outside the system.

Instead of maintaining a canonical spanning tree of shells, with all other edges distinct splices, parts of the tree could be left indeterminate until a hierarchical operation actually affects that structure. This option has not been carefully explored, but it has some attractions. Any system that maintains a canonical tree must break all symmetries in the complete graph. Any time two spaces share multiple parallel edges, one edge must be identified as part of the tree. Delaying this identification until needed permits greater symmetry. It is probable that this would reduce the number of merge conflicts (Chapter 6) that must actually be reconciled.

It is possible to give clients an explicitly different representation for splices, instead of disguising them as opaque shells. However, this puts a greater burden on agents by adding structural features and operations they must understand, and the overall functionality (thus system complexity) remains the same.

In the direction of accepting still less restricted graphs, some merge inconsistencies could be avoided by giving up trees (and even DAGs) and using general directed graphs or even the undirect dual graph as the basic structure. Communication and protection structure could carry over essentially unchanged, but a replacement for the asymmetric privileges of superior domains over inferior ones would have to be found. The basic methodology of nested abstractions would be lost, and that seems like too great a cost for too little return.

5.5.4. Implicit Composition

Instead of relaxing the strictness of the hierarchy, we could have relaxed explicit composition. For example, programming language open scope rules avoid the clutter associated with explicit configuration of all related components, especially shared access to globally exported modules.

These default rules are not appropriate for systems where configurations are to be inspected and incrementally modified during execution. Sorting through all the implicit compositions for the ones actually used by an object is not an efficient technique for determining its actual configuration. Splices (and shells) identify the interfaces actually in use by an object, not all the potential interfaces. Promiscuous splices are a convenient analog to global exports when they are really desired, while ordinary splices allow greater control over configuration.

It also seems implicit composition would complicate communication structure substantially, by the introduction of default rules for configuration, and by the extension of chains with an third form of binding between views.

Chapter 6 Partition and Consistency

6. Partition and Consistency

A well-formed HPC structure satisfies a number of constraints. For example, an endpoint may be part of at most one connection, a space must belong to exactly one domain, and the directed edges (shells) must comprise a strict tree. Under normal circumstances, HPC primitive operations preserve these constraints, transforming well-formed structures into well-formed structures. Enforced preconditions on HPC primitives prevent invocations that would result in ill-formed structure.

In general, a distributed application subject to partition and merge can not offer completely consistent service. It may provide service that is consistent over time for each client *or* service that is consistent over all clients (in a partition) at any given time. Stated another way, a distributed service must decide between discontinuous service over time or over clients. To give every client the same service, interactions with specific clients will be disrupted as previously partitioned states are reconciled and service is resumed on the basis of the new state. The service may avoid these disruptions by retaining the previously partitioned states and serving clients differently according to their differing histories. A simplifying compromise is more common. A distributed service usually maintains a single canonical state and allows access to clients in at most one partition (chosen by a quorum of resources). No matter which choice is made, the specification of a service must define the allowable inconsistencies over time and between clients.

When a partition occurs, each partition inherits the pre-partition structure, and subsequent operations can be checked for soundness within each partition. Obviously it is not possible to evaluate preconditions on structure that may have been created or modified in other partitions, and locally-sound operations may produce structure that is inconsistent between partitions. While a partition lasts, these inconsistencies are of no practical significance because (by definition of partition) they cannot be detected. When a merge occurs, however, a well-formed structure must be reestablished.

There are three basic strategies for dealing with inconsistencies due to merge. *Avoidance* restricts service so that inconsistencies at merge time are prohibited. *Reconciliation* combines several states into a single state algorithmically, possibly with a change in service. *Reporting* presents the inconsistencies to clients explicitly, permitting client resolution. All three techniques have an impact on system design.

Most distributed database consistency control techniques involve avoidance [SLR76], [BeG81], [BeG84]. Generally, the database can be written in at most one partition, ensuring the existence of exactly one authoritative "most recent" version. Avoidance is appropriate when the system does not understand the structural constraints that must be preserved (the application semantics of entries in a database), and when clients cannot tolerate asynchronous structural change (as in transaction systems). Avoidance by denying service is inappropriate for systems offering high availability, for applications that wish to apply their own consistency control policies, and for environments with a high expected frequency of partition.

The Locus file system allows updates to partitioned files in all partitions [PWC81], [WPE83]. Two forms of resolution are applied to inconsistent files. A history of updates and partitions (called a *version vector*) is used to replace inconsistent copies of arbitrary files with a dominating update, if any exists. Remaining inconsistencies in specialized files, such as mail boxes, are resolved by merging the partitioned contents of a file. Network clock synchronization protocols that maintain a distributed monotone value, with fixed upper and lower bounds on its rate of change, are good examples of complex resolution algorithms [Lam78], [Mar84], [Ray87], [WeL88]. Resolution is appropriate when the system understands the constraints that must be preserved (e.g., an update dominates a previous update on the same file, the behavior of a clock), and clients can tolerate asynchronous changes.

Resolution is not always possible, however. For example, in Locus there is not always a dominating update to a file. For arbitrary files, Locus cannot resolve such inconsistencies in a principled way, and reports them to the file owner. The inconsistent copies and their version vectors are made available so the owner can apply an arbitrary resolution policy. Locus suspends normal access to the file until the owner installs the definitive, resolved copy. Reporting moves much of the burden of dealing with inconsistency from the system to its clients. All possible system states, both ill-formed and well-formed, must be defined, the behavior of its operations must be defined on ill-formed states, and clients must have tools that can move ill-formed states closer to well-formed ones.

HPC client applications chose their own responses to partition and failure. Applications that aggressively adapt to such events can radically restructure themselves to restore lost resources and recover the desired degree of redundancy, but almost all applications will modify some aspect of process structure during a lengthy partition. HPC uses all three basic strategies to deal with the various types of inconsistent process structure that can arise during merge. The use of unique identifiers, globally known functions, and immutable properties avoids inconsistency in many HPC features. Section 6.1 shows how HPC uses these techniques, which are not specific to HPC and can be used to advantage in many distributed applications.

Applying avoidance techniques leaves a small number of structural features for reconciliation and reporting (Section 6.2). HPC reconcilation is guided by a *preservation* principle: anything that works or may be even passively of interest to an agent in any partition should be preserved in a merger involving that partition. The mathematical operation of meet on a lattice can often be used to merge partitioned structures while preserving their individual behaviors, and HPC uses several special cases of meet for reconciliation. However, some divergent structures are simply incompatible and can not be merged naturally. HPC ensures that such conflicting structures do not interfere with consistent parts of the process structure, reports the conflicts to the relevant agents, and gives agents the tools needed to reduce an inconsistent state into a consistent one.

6.1. Avoidance

Change, sharing, and partition are the major ingredients in the recipe for inconsistency. Any feature of the system that is immutable may be known uniformly throughout the system without the need for observation. In the absence of partition, all observations may be made consistent by globally simulating a single site using well known techniques, such as serialization. In the absence of sharing, all observations are trivially consistent because only one observer may look at a given piece of the system. Partition can't always be avoided (it is, after all, a failure mode), but the degree of change and sharing can be reduced as part of an avoidance strategy.

There are conflicts about privilege and authority that HPC can't merge while preserving the pre-merge behavior *and* can't trust clients to reconcile for themselves. Because HPC can neither reconcile nor report these problems, they must be avoided. By careful design, additional inconsistencies in HPC process structure can be avoided, simplifying both the (static) HPC interface to clients and the run-time merge procedures, while retaining complete availability of HPC operations during partition. These simplifications are sought for their own sake.

Despite the emphasis on dynamic change of structure, it was possible to design HPC so that most properties and structural relations are *immutable*, completely avoiding the possibility of inconsistent updates. (Appendix A gives HPC structure a formal description in terms of sets, relations, and predicates on those mathematical objects. We will refer to some of them in this Chapter.) The role, type, and structure of views, the interfaces of an abstraction, and the controller for a domain are prominent fixed properties. Less obvious examples are the pair of views that comprise a shell, and the parent of a component view. (A view's known children may change, but not its parent.)

Inconsistent changes can also be eliminated by leaving properties unconstrained, or explicitly defining sets of states to be equivalent. HPC uses at least four variations of this technique. First of all, HPC is history-less. Legal operations on a structure depend solely on its current state, and the sequence of operations that created it is both unknown and irrelevant. Second, the hierarchical relation is the only dynamic ordering in HPC. All other dynamic sets (e.g., multiplex or multicast views, shells adjacent to a space) are unordered, and all other orderings (e.g., bundle components, terminal policies) are immutable. Third, there are few upper bounds on numbers of dynamically created structures. Partitions that individually satisfy an upper bound can easily violate the bound when merged, while this cannot happen with lower bounds. Fourth, connections and spaces are unnamed, and defined by the views they relate. Making them first-class entities would increase redundancy, and the opportunities for inconsistency, without adding to the possible structures.

HPC uses three techniques to reduce sharing. Physical resources are literally tangible; they have real physical locations, and can be examined and modified only if they are in the current partition. Instead of attempting to share physical resources between partitions, HPC reports partitioning of processes and interprocess communication media via **suspended** liveness of affected communication paths. Suspension avoids inconsistencies by making the status of partitioned resources explicitly indeterminate, and therefore consistent with *any* state. The **suspended** terminal policy allows agents to generalize this behavior to multi-process abstractions that do not have any specific physical location. Third, globally unique names are created for each new piece of structure, so that similar operations in different partitions will create different structure, rather than have conflicting effects on the same structure. These unique names include the name of the generating host, so name creation does not need to be synchronized.

The visibility constraints imposed by the protection domain system also reduce sharing. First, disjoint domains drastically limit the visible effects of an operation. Only the **new** operation and promiscuous splices have direct effects in two domains; the liveness property also propagates across domain boundaries. Second, the HPC system is free to hide information from its clients, especially non-hierarchical structure. If the exact relations between domains were visible to clients, the facade of a strict hierarchy could not be maintained.

6.1.1. Domain Contents

The contents of domains are formally described by the relation member(v, d), where structural element v is a member of domain d. (Because shells have been equated to pairs of views, and both spaces and connections can be formally defined in terms of views, we can treat views as the only kind of structural element.) It is a critical property of the protection system that any view is a member of at most one domain. There is no way to express membership in multiple domains, there is no way to designate an intelligent, neutral party to arbitrate conflicts between agents of different domains, and the cooperation of agents cannot be assumed in matters as crucial as protection, control, and authorization. For these reasons, membership in multiple domains must be avoided.

During a partition, one agent for a domain old that existed before the partition started could invest a new domain new in a subtree containing view v. Suppose that v is moved from old to new. No matter what agents in other partitions do, when the partitions are merged together, v will belong to new and at least one other domain. If they do nothing to v, it will belong to old. If they create a different new domain through another invocation of invest, v will belong to the new domain. (There is no way agents in different partitions can independently create the same new domain.)

These violations of the constraint are avoided by making the domain of a view an immutable property fixed when the view is created. When domain creation and destruction transfer structure between domains, a distinct *copy* is created in the new domain, isomorphic to the affected structure, then the original structure is deleted. Another interpretation is that the affected structure is *renamed*, because the HPC system's internal representations of abstract structure can be modified in place.

An object seldom will be an agent for both the old and the new domains. When only the old or the new is visible, the exact isomorphism between them is not critical. However, there will be occasions when the correspondence is useful or necessary. It can be easily preserved by structuring the name space. An HPC unique identifier consists of a (unique) domain field and a (unique) element field. When a view is renamed, only the domain field is altered. This also reduces the consumption of unique name space, as only one new name is required to rename arbitrary subtrees.

6.1.2. Characteristics and Immutable Relations

Renaming is a basic technique for treating a fixed set of things with dynamic properties as a dynamic set of things with fixed properties. It avoids fatal inconsistencies (such as conflicts about privileges) by introducing less dangerous ones. For example, a view may exist in some partitions but not in others. However, creating and destroying views produces exactly the same effect, so renaming does not introduce a new problem. HPC's preservation principle resolves presence/absence conflicts during merge by retaining views present in any merged partition. Specifically, that means that a view that was deleted (or renamed) in one partition may be resurrected in a later merge.

Creation, destruction, and renaming of views and related structure within the current partition are dynamic operations, but HPC formal consistency is simplified by treating the set of views and the relationships among them as immutable. The technique of *characteristics* and *immutable relations* used in this formal treatment is not specific to HPC and can be applied to many data structures that are dynamically created and destroyed but otherwise have fixed properties throughout their lifetimes.

The HPC process structure in the current partition is a subset of the structure in entire environment. The local structure is known exactly, but the non-local structure, by definition, can't be known, so we can assume anything about the global structure that is consistent with the local structure as a subset. Specifically, we assume the global structure is immutable, and define the structure within a partition as the intersection of the fixed global structure with a dynamic local characteristic function (or set) that defines the views and other features known in the partition. As views are created and destroyed, they are added to and removed from the local characteristic, but the global structure remains unchanged.

When this definition can be applied, an especially simple merge procedure is possible. Partitions can be merged by taking the set union of the local characteristics and the local structures. This preserves the defined relationship among the resulting characteristic, local structure, and global structure. A proof of the consistency of this merge procedure, and other advantages of the technique, are given in Section A.1.

HPC structural features that have fixed properties throughout their lifetimes can be defined this way. The role, type, and structure of a view are such fixed properties, as is domain membership. Let the characteristic c-view(v) define the views known in the current partition, and let c-member(v, d) define the domains they belong to. As each view is created, it is added to the local c-view, and its permanent domain is added to the local c-member. When a view is destroyed, it is removed from c-view and c-member. The global member(v, d) is a similar, immutable relation defined for *all* views, *all* time, and *all* partitions. By defining c-member as the intersection of c-view and member, we are assured that taking the union of several partitions' c-view's and c-member's is consistent with the definition. Writing down member(v, d) would require full knowledge about the future, but we don't actually have to compute it. Instead, it can be "virtual": The identifier or domain of a view is never needed until it is created or made available through merge. Destroying a view simply removes it from the known *local* structure. If it has not been removed from all partitions, it may become known again after a merge, consistent with HPC's preservation principle. Other fixed view properties are treated the same way.

View hierarchies are a more interesting example, because the components of a view may change dynamically. The characteristic is again c-view, while the relation component(c, p) holds when c is a component view of its parent p. Here, a technical constraint on the arguments of the global relations is very important. When we create a view we certainly know its ancestors, but certainly not all its (as-yet uncreated) descendants. Therefore, we will insist that the second argument of a local relation is characteristic whenever the first argument is characteristic.[10] For example, the apparently equivalent parent(p, c) can *not* be used as a local relation, because we can not know all future descendants of any currently known view.

Use of the characteristic technique interacts with other HPC design options, e.g., if deleteing a view caused its components to be inherited by its parent rather than destroyed, the global component relation could not be immutable. The bound relation that defines direct private peer bindings (from shells and splices) is constrained similarly. When one view of a shell is renamed, both views must be renamed to keep bound immutable and single-valued. That means that opaque domain boundaries may "spontaneously" rename even when the domain on the visible side is unchanged. This in turn implies that the domain field of a structured HPC identifier does not suffice when renaming domain boundaries.

6.1.3. Splices

As described previously, the splice operation is a cooperative, two-step operation. In a centralized environment this offers no difficulty, but there are many opportunities for confusion in a partitionable environment. The partial ordering of distributed events in separate partitions can make "first" and "second" steps undefinable. Inconsistent steps can be taken in different partitions, which are exposed at a later merge. Different numbers of consistent steps can be taken in different partitions, causing inconsistent merges. Because the other HPC operations are one-step and effectively atomic within a single domain, the technique used to avoid inconsistencies with splice merits a detailed examination.

[10] Relations with this property are sometimes called *serial*.

Both first and second splice steps take an empty shell and create a new domain boundary. Call the local and remote shells in the first step old-L and old-R, respectively. To keep track of associated first and second steps, there is an implicit relation between the old and new views. Let prespliced(old-L, old-R, new-R) relate the arguments of splice(old-L, old-R) with the hidden view created to replace old-R. An invocation of splice will execute the first step if there is no tuple with its arguments reversed in the prespliced relation, and complete the splice using the hidden new-R otherwise. To be well-behaved, prespliced must identify a unique new view for a given pair of old views, and identify no view if the old views have not been prespliced.

Without restrictions, partition will violate these constraints. A first step carried out in two partitions could create both prespliced(old-L, old-R, X) and prespliced(old-L, old-R, Y). After a merge, will **splice(old-R, old-L)** complete a splice using X, Y, or both? A first step carried out in only one partition could lead, after a merge, to prespliced(old-L, old-R, X) at the same time that old-L is in the partition. Will **splice(old-R, old-L)** complete a splice using X or take the first step in splicing to old-L? The likelihood that agents for *both* ends carried out their splice while partitioned complicates this scenario, making the operations first steps in opposite directions, rather than a first and a second step. Do two opposing first steps complete a splice?

Fiat removes some of the ambiguity. In HPC a prespliced shell takes precedence over an unspliced shell in a second step, by definition. However, multiple first steps in one direction, and opposing first steps, remain a problem.

A partitioned second step following a first step that establishes prespliced(old-L, old-R, X) is less awkward. We can determine the identifiers for the complete splice bound(new-L, new-R) during either the first or second steps, with differing results. The first step can establish X as new-L, and fix the binding between new-L and new-R even though it is not used until the second step, which would always replace old-R with new-R. Identically named structure is identical, so this produces one splice, even when the second step is executed in multiple partitions. The alternative is to determine new-R during the second step. The first step would replace old-L with X, which would be bound to some hidden view, and the second step would replace X with new-L at the same time that old-R is replaced with new-R. Executing the second step in multiple partitions would produce distinct splices because each partition would select unique values for new-L and new-R. The effect is as if both steps had been executed while partitioned.

There are three general ways to avoid the remaining inconsistencies in the prespliced relation: enforce a centralized decision, enforce a consistent distributed decision, and accept lower standards of consistency. Four specific mechanisms were considered for the HPC design.

- Synchronous splice

If the two steps in splicing were **splice(old-L, old-R); splice(old-R, new-L)**, the operation would be synchronous, delaying the second step until the results of the first step are available. The agent taking the second step explicitly resolves any ambiguity. Besides introducing asymmetry to the agents negotiating the splice, this mechanism adds another step to the negotiation, because the first agent must communicate new-L to the second.

- Issue a ticket

Another way to enforce a centralized decision would be to ask the HPC agent to precompute new-L and new-R and issue a ticket that associates all four affected views. Splice would then take a ticket and the local view as arguments rather than two views. This would require another basic operation for ticket creation, but has some attractions. While the ticket buyer must know both views to be spliced, the ticket user need only know the view in its domain.

This adds a little modularity. The implicit relation of ticket 4–tuples is obviously an immutable (virtual) relation.

It is tempting to use tickets for splices that are not tied to specific target views. Tickets could be created independently of the views to be spliced, and passed from agent to agent. However, like other capability schemes, there would be a insoluble use-once problem in a partitionable, fully-available environment

- Accept multiple private bindings

The acceptable structures could be extended to allow a second step to splice a view to more than one other view at a time. There is nothing fatal about allowing a view to be bound to more than one view, although we chose not to do so. The effect of multiple private peers on communication patterns is well-defined: multicasting. Multiple binding might also serve as a way to accept the effects of multiple ticket uses. However, these multiple bindings would be completely hidden, unlike the multicasting introduced by complex views. This seems undesirable, especially if some media may not be multicast.

- Known function on structured names

Distributed agents will reach consistent conclusions if they apply the same deterministic algorithm to the same data. A pairing function globally and consistently can compute unique identifiers (new-L, new-R) as a function of (old-L, old-R). Appending the old names or interleaving their bits are typical pairing functions. However, a pairing function requires some structure in the name space to avoid generating "fresh' identifiers that might be given to a completely new structure. Also, a paired number has all the bits of its inputs, so the name space must be big enough to contain the largest result. These conditions are met in HPC. A (pre)spliced shell can not be spliced again since it is not an empty shell, so a pairing function will be applied only to fresh names. (When the splice is removed, a completely fresh name is assigned to the empty shell.)

Despite the sparse use of name space, HPC uses this technique because it is fundamentally distributed and preserves the desired symmetry and asynchrony of the splice operation. It also has the side-effect of creating one splice, even when the affected views are spliced in multiple domains, or when the both ends are spliced in separate partitions. The ticket mechanism also has these nice effects if a single ticket is used, and avoids wasting any name space.

6.2. Reconciliation and Reporting

Through careful design, most dynamic aspects of HPC process structure can be treated as partial local knowledge about immutable global structure, using the technique of characteristics. These structural features can be reconciled upon merge simply by taking the set union of the data in each partition. Set union is one specific function in the general class of *meets* over *lattices* of values (vid., Chapter 6 of [Sto77]). The meet operation has some highly desirable properties for reconciliation. It is stable, idempotent, and convergent; merging any number of partitions any number of times in any order produces the same ultimate result. The preservation principle we adopted to guide reconciliation can be obtained by careful choice of lattice. Max and min are other examples of meets over appropriate lattices.

There are only three relations in the formal description of HPC structure that can not be handled by characteristics. Merges of these features are handled by specialized meets for reconciliation, with explicit reporting of cases where a structure-preserving meet could not be chosen. policy(d, type, p), defines the temporary and permanent terminal policies for a domain. Connections are defined by the symmetric relation connected(v1, v2), and spaces are defined by the equivalence classes of the relation adjacent(v1, v2), which identifies pairs of views incident on a common space.

6.2.1. Terminal Policy

There can be only one terminal policy of each type for a domain. Within a partition, the most recently specified policy is used, but different policies can be specified in different partitions. (Once again, partial ordering of events in different partitions means that the "last" modification is not defined during a merge.) Inconsistencies in policy cannot be reported to the client, because the policy specifies what to do when there is no client to report to. A merger may continue a temporary loss of control or reveal a permanent loss of control, so a well defined policy must be available immediately.

Reconciliation uses a non-trivial example of meet over a lattice. Because HPC understands what terminal policies mean, it can resolve inconsistencies according to a sensible set of priorities. For example, **suspend** prevents unauthorized interactions, while **null** does not, and **die** hides structure from the outside domain, while **abdicate** does not. Our priorities in descending order are to conceal structure, exert control, and keep running. Accordingly, the basic policies are partially ordered:

suspend/animate < null < die < abdicate,

where **suspend** is incomparable with **animate**, and **animates** with different parameters are incomparable with each other. Policy sequences are partially ordered lexicographically, taking the default basic policy as the first element (the reverse of the order in which basic policies are applied.) For example:

die < die suspend

abdicate suspend < abdicate die

... animate(x) is incomparable with **... animate(y)**

... animate(z) < ... null

The meet on this lattice is the greatest element less than or equal to its inputs. For example, the meet of:

die suspend animate(x)

die suspend animate(y)

die abdicate

is **die suspend**. The effect is to take the longest common prefix of incomparable sequences, and keep the best, according to our priorities, of the sequences that remain. Since HPC applies one basic policy at a time until it gets to one that works, the interpretation of incomparable policies is that the conflicting portions don't work.

There are other reconciliation strategies worth consideration. The basic policies can be ordered differently, according to different priorities, and lexicographic ordering is not the only way to compare sequences. For example, none of the basic policies in a sequence up to the last (first applied) non-**animate** will ever be tried (because a non-**animate** policy never fails), so this prefix could be discarded as irrelevant.

6.2.2. Connections

The connected relation can not be modelled as the intersection of a characteristic with an immutable relation, because connections are created and destroyed without creating and destroying the views they join. A view may have at most one connection, and this constraint can obviously be violated upon merge. HPC cannot resolve the inconsistency without making an arbitrary choice of connection(s) to remove and this would violate the basic principle of retaining all structure from all partitions, so the meet/lattice technique was not used.

Instead, the HPC system reports this inconsistency to the agents of the domain to which a multiply connected view belongs. Undefined behavior is avoided by **suspending** the view, and all communication paths passing through it, until a legal number of connections is restored. Agents can use the **disconnect** primitive to remove extra connections as easily as legal ones, so the tools needed for user reconciliation already exist.

There are two implications for system design. First, the format of public peer notifications must allow arbitrary numbers of peers even though at most one is normally permitted. Second, the **connect** and **disconnect** primitives cannot be exact inverses, because the preconditions for **disconnect** must be more general than the postconditions that **connect** can establish. This asymmetry could be removed by allowing **connect** to create more than one connection to a view, automatically suspending it.

6.2.3. Space Hierarchy

The proper nesting of objects is the most difficult structural invariant to restore during a merge, and the dual graph representation is essential when expressing the possible inconsistencies and their reconciliations. Since **enclose** and **disclose** modify the dual graph without creating and destroying all the views of the affected spaces, a representation for spaces is a fundamentally mutable relation. Two formal relations define the dual graph: `bound(v1, v2)`, which describes the immutable edges (shells and splices), and a mutable relation, `adjacent(v1, v2)`, which indirectly describes spaces by their incident views. The pairs of views incident on a common space satisfy `adjacent`, so the complete set of views incident on a space is an equivalence class of the relation. This indirect description is technically more convenient than the obvious relation between views and explicit spaces `incident(v, s)`.

Merge can transform a set of strict trees into an arbitrary directed graph. This is the primary difficulty in reconciling the object hierarchy. Figure 6.1 shows the merger of a partition with three nested shells, *A*, *B*, and *C*, and a partition in which *B* has been **disclosed**. (The arrows shown on directed edges point away from the root.)

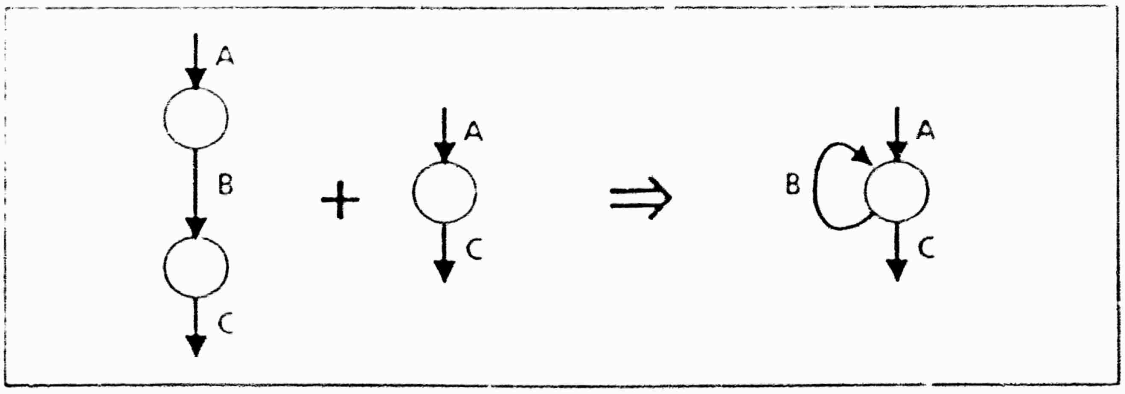

Figure 6.1. Loop in the Dual Graph

Figure 6.2 shows two partitions in which **enclose(B)** has been executed on a common initial structure of nested shells, *A* and *B*, and their merger.

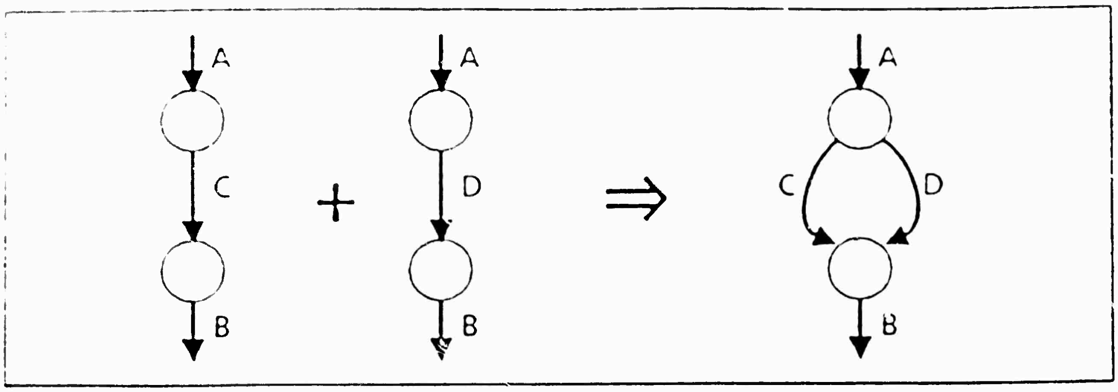

Figure 6.2. Parallel Edges in the Dual Graph

More complicated examples with larger loops and incomparable branches can easily be constructed.

The violations of the strict hierarchy can be removed by converting some of the directed, non-tree edges into undirected splices. For example, the simple loop of Figure 6.1 can be converted as shown in Figure 6.3.

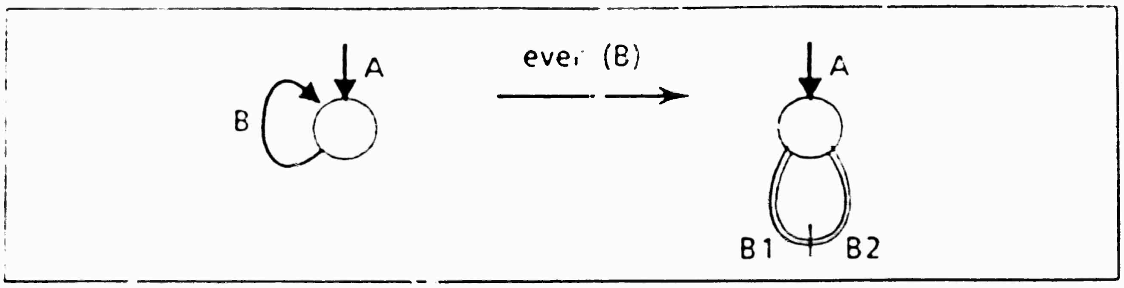

Figure 6.3. Breaking Loops through Eversion

The hierarchical view of the pre-merge and post-conversion structures (Figure 6.4) shows that shell *B* is effectively turned inside out, with one end of the opaque splice (*B1*) representing the exterior of *B*, and the other end (*B2*) representing its interior. Because of this effect, we call the conversion technique ***eversion***.

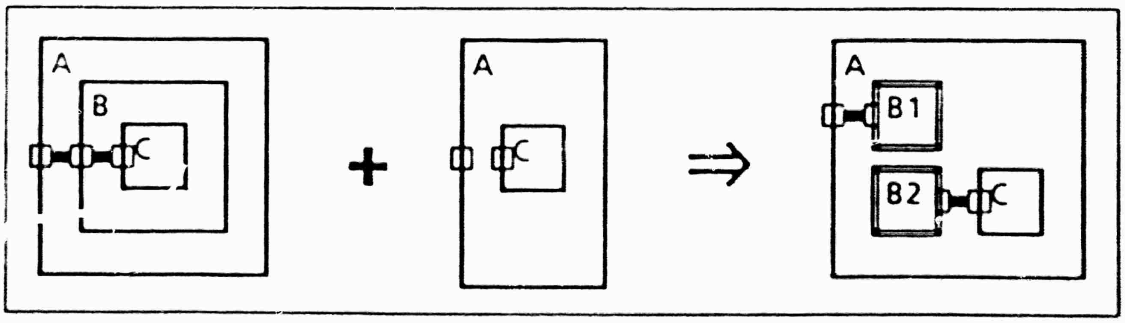

Figure 6.4. Hierarchical View of Eversion

We can now describe the reconciliation of the hierarchy within a single domain under the assumption that the domain has a unique root. The procedure is as follows:

(1) Find (transparent, directed) shells that have been converted into (opaque, undirected) splice format in some partition, and evert them in the merger

(2) Take the transitive closure of the union of the local `adjacent` relations. In general this will merge spaces and give spaces multiple parents.

(3) Compute immediate dominators and back-edges in a single pass over the dual graph. Break all loops by everting back-edge shells.

(4) If a space has more than one incoming shell, evert them all, and merge the space with its immediate dominator.

Spaces with incident edges `v1` and `v2` are merged by adding `(v1, v2)` and `(v2, v1)` to `adjacent` and taking the transitive closure of the relation. This closure step may be performed once, after all spaces have been processed.

Any number of partitions can be merged in any order, and the eversions and space merges during a partition merge can be performed in any order and still yield a unique result. This procedure hides all non-hierarchical structure behind splices. Equally important, it preserves all compositions, and therefore all behavior. It does so by modifying the hierarchy in ways that clients can not, so this automatic reconciliation is not transparent. Unlike renaming, eversion can not be disguised as asynchronous behavior by another client agent.

The root domain, which consists of one space with incident edges for the top-level objects, requires slightly different treatment. Because there is no root shell, and because the top-level objects in different partitions might be disjoint, the root space may be represented by several equivalence classes of `adjacent` instead of one. The problem is avoided simply by distinguishing the root space.

The first step given above, everting shells that have already been everted in a previous merge, could be omitted and still result in a consistent object hierarchy. However, the reconciliation procedure would no longer give the same result for merges of partitions in different orders. As with splicing, eversion offers a choice of creating new splices every time any shell must be converted, or a single splice for a given shell, no matter how many times the shell was converted in previous merges. We chose to create a single splice to reduce the amount of structure created during merge, and use a pairing function to unambiguously relate a shell and its post-eversion splice.

Earlier reports described a mechanism for reporting and user reconciliation of inconsistencies in the object hierarchy (**detach**) [LeF85], [LeF85], but the full implications on HPC system design were not understood at that time. Both system and agent complexity are reduced by automatic reconciliation of the directed tree.

6.2.4. Domain Hierarchy

Now consider domains with multiple superior domains. There may be no unique root within the domain to base intra-domain reconciliation on, and operations like **abdicate/depose** and **die/kill** are no longer well-defined, because there is no unique superior domain boundary or well-defined subtree to remove. Partition and merge can easily produce such situations. Starting with two pre-partition domains `above` and `below`, during a partition an agent for `above` can invest control of the subtree containing `below` to a new domain, `middle`. In one partition, `below`'s superior domain is `above`, in the other it is `middle`. More complex sequences of domain and space operations can lead to structures as shown in Figure 6.5. (Domain boundaries are shown with double lines.)

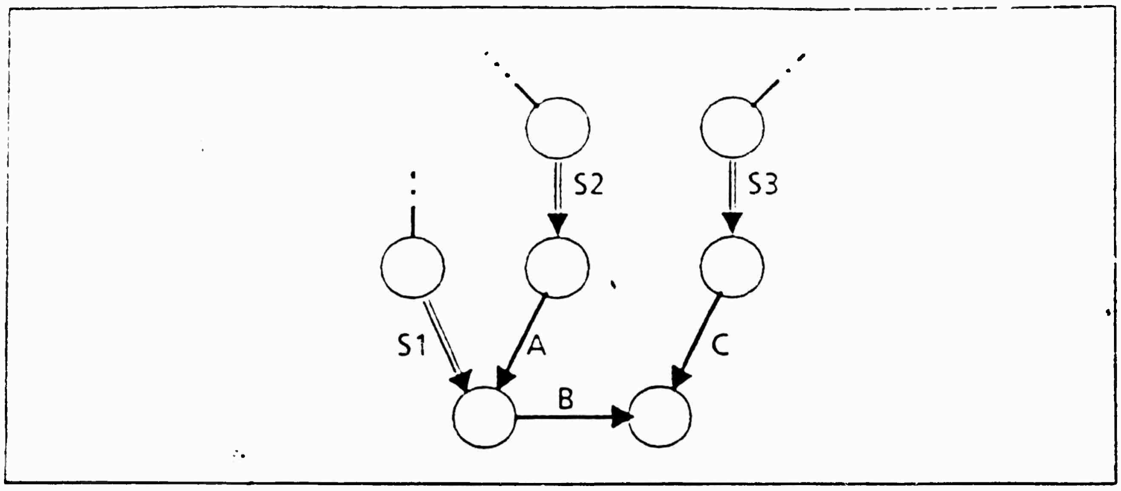

Figure 6.5. Multiple Superior Domains

Each domain is prepared for intra-domain reconciliation by merging all the spaces with an incident superior domain boundary before reconciling the domain internally. This ensures a unique root space within the domain, makes all the superior domain boundaries incident to the same space, and brings each of the partitioned domain roots up to the root of the merged domain. Figure 6.6 shows this step applied to the structure of Figure 6.5.

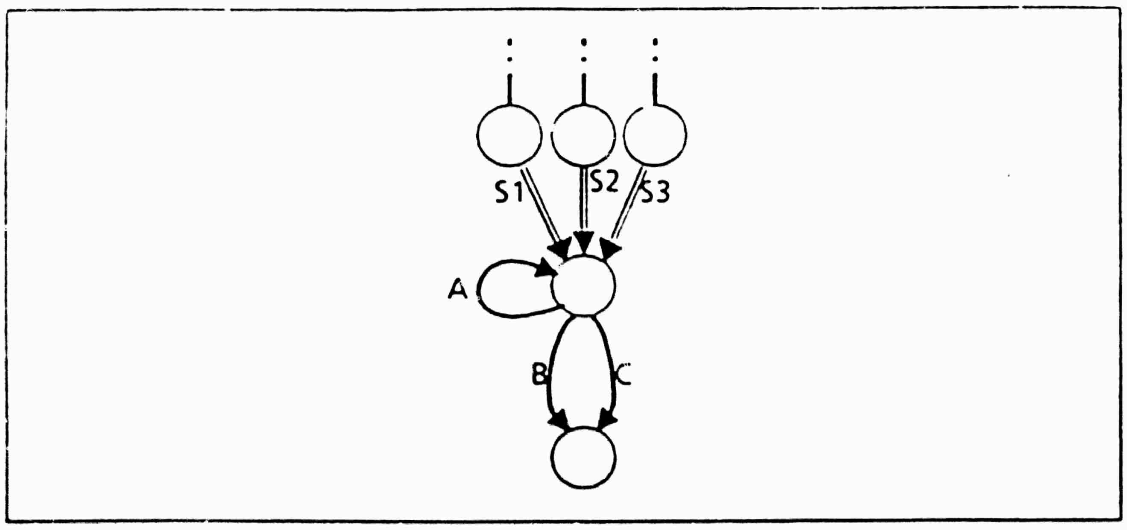

Figure 6.6. Superiors Isolated at the Root

The remaining inconsistency of multiple superiors to a domain is treated similarly as multiple connections to a view. There is no function-preserving transformation that can be automatically applied, so the multiple superior boundary views are reported to the agent of the inferior domain, and forced suspended. Because only the *superior* domain boundaries are suspended, the domain agent process(es) are able to coordinate and execute a response (but see below.)

To allow clients to reconcile this kind of inconsistency, the definitions of abdicate/depose and die/kill are extended to excess superior domain boundaries. Inside the domain, an excess domain boundary is simply removed. Outside the domain, the domain is replaced by an empty leaf. When only one superior domain boundary remains, it is unsuspended and the domain operations will have their usual effects. This is externally consistent with the behavior of depose on a domain suspended because its agents are temporarily partitioned. An implication is that kill on a given shell only destroys the structure it dominates. For DAGs rather than trees, this may be strictly less than the structure it precedes in the hierarchy.

Terminal policies of abdicate and die are extended to excess superior domain boundaries simply by applying them to all superior boundaries rather than the usual unique boundary that is implicit in the policy. All boundaries will be treated as excess and the domain will be effectively killed.

While this strategy leads to DAGs, rather than trees of domains, the graph of domains still behaves like a tree. Domains can interact through communication or domain operations only when the subgraph relating them is a tree.

As a simplification of the HPC interface to processes, simple domains are treated specially to ensure a single superior domain. Because processes are physical resources, they belong to a single, well-defined partition (unlike abstract complex objects). If a merge gives a process multiple superior domains, the boundary known in the partition of the physical process is retained, and the boundaries known in other partitions are replaced with empty leaves. During the partition, these other boundaries will be suspended, representing uncertainty about the process's continued activity, so this treatment, like renaming, is masked as asynchronous behavior of other agents.

Because structure is reported to clients strictly in the hierarchical view, the format for reporting a shell's parent must allow arbitrary numbers of parents even though only immediate children of the superior domain boundaries will have multiple parents. Similarly to the case of multiple connections, inverse domain operations are not exact inverses, but the asymmetry can not be as easily removed in this case because there is no operation to create a directed domain boundary between arbitrary domains.

There is a further, subtle, interaction with protection structure and the system interface. It is possible, if unlikely, for all paths to the domain's controller to pass through one or more of the superior domain boundaries, leading to a temporary loss of control when the paths are suspended. In this situation, an external agent can not restore control directly, it can only give up control by destroying one of the surplus domain boundaries. When all but one of boundaries have been destroyed, control may be restored via paths through the remaining boundary. Because the (possibly many) external agents have no way to coordinate their actions, restoration of control is unlikely. Even "generous" agents willing to yield control can get into trouble because of the asynchrony in the system interface. An agent could destroy the final boundary before learning the suspension had been lifted.

We declined the aggressive investigation of more general structures that this interaction suggests. We could accept multiple superior domain boundaries as well-formed structure, but would prefer an explicit operation to create such boundaries instead of making it an exclusive side-effect of uncontrollable partition and merge. Alternatively, the hierarchical relation between domains could be done away with, so that all domain boundaries are undirected splices, but the problem of destroying another domain, and any auxiliary domains it created, remains. Authorization to destroy a domain, without knowing or controlling its internal structure, must be expressed somehow, if not hierarchically.

Chapter 7 Prototype Implementation

7. Prototype Implementation

At a distance, the HPC system is a three layer cake. At the bottom is a collection of host operating systems, at the top is a collection of client processes, and in between is HPC software. The middle layer consists of three types of processes: kernel, host IPC router, and host process manager. The kernel maintains the database of abstract structure, and determines the resources needed to implement an abstract operation.

The client-HPC interface provides agent processes access to abstract structure like shells and interfaces, and worker processes the host IPC resources needed for end-to-end communication. Clients interact with the HPC kernel using a complex application protocol on top of standard network connections (TCP/IP). (Section 7.1.)

The host-HPC interface deals with physical resources like IPC media and processes. The router and manager processes isolate host dependencies of resource creation, destruction, and status monitoring from the rest of the software. They are the only system components to communicate directly with hosts. Sections 7.2 and 7.3 discuss their internal structures and interactions with the kernel.

The kernel process is the heart of the system. Section 7.4 describes the basic software packages that handle interactions with the outside world and maintain the internal database of abstract structure. Incremental updating of global connectivity as a result of local abstract operations is an important function of the database.

The final Section presents some experiences with essential, desirable, or inadequate tools and programming support.

7.1. Client Interface

The client-kernel interface has three functions: integrate new clients into the HPC system, implement the interactions between agent clients and the HPC kernel, and provide worker clients with the real IPC capabilities represented by abstract endpoints. These functions are summarized here. For a full description of the C language/UNIX operating system interface binding and the underlying network application protocol, see [Fri86].

7.1.1. Registration

Client processes are integrated into the HPC system by two TCP/IP connections. Clients connect to a well-known kernel port to *register* with the kernel. The first step of the registration protocol creates the second connection.

The client sends its host and process numbers to the kernel, which determines if the client has been animated. If not, the kernel creates a new shell with no interfaces as an immediate child of the HPC root. This is how new independent applications enter the system. After locating or creating the appropriate shell, the kernel sends its interface descriptions to the client.

If the client is implementing a complex domain, it describes the interfaces it wants on the new shell to be created, and specifies the interface to be connected to the new controller. The kernel acknowledges. For both simple and complex domains, the client sends an explicit end to the registration protocol.

7.1.2. Agents

Interactions between agents and HPC use both client-kernel connections. One connection is used in a synchronous, bidirectional protocol. For every invocation of an HPC primitive, the client sends a rigidly formatted message. The HPC kernel responds immediately with a message containing a globally unique "request" number that will be used later to refer to the invocation.

The kernel uses the second connection asynchronously and unidirectionally to send the client three types of information messages: invocation error reports, notifications of structural change, and responses to the inquire operation. Every message follows a rigid format, and begins with the request number for the invocation that ultimately prompted it. For messages triggered by events outside the HPC system (e.g., failures) the request number is a distinguished value.

Every time a structural element is created, destroyed, or modified, a notification is sent to all agents for the affected domain. This is the response to successful invocations of HPC primitives. One invocation may generate many notifications carrying the same tag. These notifications are copious and brief to eliminate the need for polling by agents.

- An enclose operation generates a creation message for the new shell, creation messages for every one of its interface views, change messages for its children, and a change message for its parent. A disclose operation generates a deletion message for the shell, deletion messages for every one of its interface views, change messages for its children, and a change message for its parent.
- When a new interface component is created a creation message for the new view, and a change message for its parent are created. If the new view is complex (e.g., a bundle) creation messages are generated for all its components.
- When liveness or connectivity changes on a view, a change message is generated.
- When a shell becomes a domain boundary, deletion messages for all the previously visible structure are generated. When a domain boundary is dissolved, creation messages for all the previously invisible structure are generated. In both cases, a change message for the shell is generated.

7.1.3. IPC Terminals

Real communication between worker processes requires translation of abstract simple endpoints inside simple domains into host-specific I/O facilities accessible from the worker processes. Worker processes then use host I/O operations to communicate with one another.

An IPC *terminal* is a host I/O handle (e.g., UNIX file descriptor) together with any additional resources needed for the HPC system to connect and disconnect workers. Before a client can use an endpoint, it must be translated into a terminal. Clients must explicitly begin translation to avoid unnecessary consumption of per-host and per-process resources. Clients can reclaim terminal resources by destroying the terminal.

Creation and destruction of a terminal has no effect on the abstract endpoint except that liveness reflects connectivity to real terminals. Roughly, terminals are to endpoints as physical pages are to virtual pages, except that clients must do their own resource management.

To associate the I/O handle with the endpoint, and to create any other needed resources, the client and kernel use a synchronous protocol over the first connection. In this implementation, terminal creation requires an exchange of five messages. This is only part of the protocols needed for terminal manipulations. The kernel is engaging in a similar, but more complex, protocol with the IPC router process at the same time.

The client interface library hides the terminal protocols, just as it hides the invocation protocol, and so prevents client programming errors.

7.1.4. Discrepancies

The client-kernel interface was frozen early in the development of HPC, leading to some discrepancies between design and implementation. The most glaring discrepancy involves control messages between agents and controllers. As designed, an agent process wishing to invoke some operation sends a message to the controller of domain to be affected, using a control interface. The usual rules about structural compatibility and end-to connections apply. A controller's multicast interface ensures that all agents connected to a controller receive its notifications and that the controller receives messages from all agents.

As currently implemented, an agent process always invokes HPC operations, on *any* domain for which it has privileges, through the procedures provided in the application interface library. The agent does *not* send messages to connected controllers. However, it *must* have connections to the appropriate controllers. An agent's connections determine its privileges, just as if control messages were being sent. If an end-to-end connection to the appropriate controller exists, the kernel carries out the operation.

The interface library sends messages directly to the HPC kernel using the invocation protocol, so control messages can not be intercepted, filtered or debugged as described in Chapter 3. Also, agents can not separate control activities for separate domains onto separate local interfaces. Sending invocations along connections primarily associated with an agent, rather than a domain, also led to a distortion inside the HPC kernel, that will be discussed later.

In this centralized implementation, there will never be merge inconsistencies. Since no domain will have shells to multiple parents, the arguments to **abdicate** and **die** are implicit.

The kernel administers both temporary and permanent policies, but the interface only allows a request for a new permanent policy consisting of a single basic policy (**die** or **abdicate**). Clients can not request alternatives to the default temporary policy.

Endpoint/extension promotion is the final omission. Worker clients are allowed to translate either endpoints or extensions of simple structure into IPC terminals, so promotion is implicitly allowed for simple domains. However, there is no provision for promotion in complex domains. The details of managing a multi-agent interaction that must take place immediately after a new domain is created, before any other operations are invoked, and affect every view at most once, were never worked out.

7.2. IPC Router

To translate abstract connectivity into real transport connections, the HPC system must provide clients access to host IPC resources, and dynamically reconfigure the transport connections between clients. Unfortunately, most host systems and standard protocol suites lack *third-party connect*, a simple facility that would make reconfiguration a trivial matter.

Lacking third-party connect, dynamic reconfiguration is complex and expensive enough to justify isolating host-specific IPC functions in a separate IPC router process. The router supports creation and destruction of client terminals, creation and destruction of end-to-end connections, and additional diagnostic functions. A TCP/IP connection is used with an asynchronous, highly multiplexed, application protocol for command functions between the kernel and the router.

7.2.1. Third Party Connect

Ideally, the HPC kernel could look up the IPC terminals for the endpoints of an end-to-end chain, and instruct the host operating system to set up or tear down a transport connection between the terminals. The kernel would control reconfiguration, connecting and disconnecting arbitrary pairs of processes (and terminals), without the participation, or even cooperation, of the affected clients. This scenario preserves the distinction between communication (workers' responsibility), configuration (agents' responsibility), and implementation (kernel's responsibility).

Real operating systems and protocol suites place unfortunate and, we argue, unnecessary constraints on the creator of the transport connection and the connected processes, leading to intrusive, unsafe, or inefficient (or all three) emulations of the desired third-party connect property.

For example, a UNIX pipe must be created before the processes it connects, by a process that creates the connected processes. This does not allow reconfiguration of existing processes, and is totally inadequate to support HPC.

Most operating systems with more sophisticated IPC objects (Charlotte links, Accent/Mach ports, 4.3BSD pipes, etc.) permit passing a link end along an existing link. However, clients must actively participate in reconfiguration, by continually monitoring a link to the HPC kernel for messages containing new links, discarding old terminals for a given endpoint, and installing new links as terminals. There is no way to enforce, or even inspect, that clients act correctly. They could retain old links (in systems where the kernel can not retain destroy rights), ignore new links, or send links to other clients, bypassing agents and the kernel entirely.

Standard network protocol suites such as IP provide even less support for transparent reconfiguration because the kernel must give a client process full details about the address and identity of its peer before a connection can be created.

Third-party connect is intrinsically inexpensive. For example, TCP/IP uses a well-defined sub-protocol for connection set up and tear down, with clean interactions with the main protocol governing reliable delivery and flow control. If a third process could initiate this sub-protocol across the network, instead of the two communicating processes from the host interface, we would have third party connect.

A principal reason that link systems and network protocols restrict configuration as they do seems to be authentication of authorized configuring agents, alternatively, equation of protection with holding a link or connection (capabilities). However, even the absence of a trusted global authentication system does not stand in the way of a practical third-party connect mechanism. To continue the TCP/IP discussion, each client could specify the host and port from which authorized configuration messages are allowed. This provides third-party connect with the exactly same degree of security as use of unmodified TCP/IP. We discuss this issue further in [Fri87].

7.2.2. Forwarding

In the absence of third-party connect, and a notable (two and a half year) delay from our workstation vendor in providing sufficient sources to build in the desired support, we chose to provide an unintrusive, secure, but inefficient emulation for the UDP and TCP protocols. When a client IPC terminal is created, a fixed connection is made to a similar terminal owned by the IPC router process. Reconfiguration is performed entirely inside the IPC router. Clients are never requested to modify their terminals in any way, nor are they given any information about their peers.

During the client-kernel protocol that translates an endpoint into a terminal, a similar router-kernel protocol is executed to create a router terminal and connect it to the client. The primary difference is that the router-kernel setup sub-protocol may be multiplexed among concurrent exchanges between the router and kernel.

After terminal creation, the router keeps track of which host terminals correspond to which endpoints. When end-to-end connections are created and destroyed, the kernel instructs the router to start and stop forwarding messages between the endpoints. Therefore, connected workers do not send data directly to one another, but first to the router process, then on to the destination.

Besides the obvious inefficiencies of sending every message twice, the router is a bottleneck. Even in this prototype, HPC system and client processes can be freely distributed. Messages that might be processed in true concurrency are serialized in the router. Executing router functions in a separate process, and not inside the kernel process, adds unwanted latency to handling both agent invocations and worker communications. Providing one router for each physical processor (HPC does not assume a single router) might avoid the communication bottleneck, but further problems arise: finding the router for an endpoint, potential router-to-router forwarding inefficiencies, etc.

The IPC router setup does have some advantages, of course. Besides client simplicity and security, it offers an obvious experimental implementation for multicasting media such as TCP, which don't have native multicast semantics. For each terminal, the router keeps a list of all the other terminals to receive outgoing copies of incoming messages. For a normal connection, this list has just one terminal. (This prototype does not attempt to handle pathological interactions of limited buffering and reliable retransmission.)

7.2.3. Implementation

Because IPC is *the* fundamental issue in real interactions between objects, the router and the router-kernel protocol were the first components of the HPC system to be implemented. They were fully, and easily, debugged with extensive scaffolding processes standing in for clients and the kernel. The success of this approach prompted similar unit-and-interface test procedures for the other components. Full details of the implementation are given in [FrP86].

The router is a very highly multiplexed server, and care was taken that it would never block under any circumstances. For each output terminal, it must first wait for available operating system buffer space, then wait for input on the corresponding input terminal, read the input without blocking, write the output without blocking, internally buffer any excess that could not be written, and deal with operating system errors or resource limitations at any point. Of course, many terminals are bidirectional so, this is done in both directions, and multicasting further complicates things.

At the same time, the router is engaging in overlapped, non-trivial protocols with the kernel to handle command functions. It must be ready to receive a new message from the kernel at any time, determine which in-progress operation it applies to, and advance the operation the corresponding step. Unique tags are used in the router-kernel protocol, as in the client-kernel protocol to simplify demultiplexing.

The router is based on a table driven state machine. Each terminal has an entry in the table to store the state of its submachine. An incoming event on a terminal recovers the previous state from the terminal, then calls a function to do the next step. This was an adequate technique, but we would never use it again. The lightweight tasking package (developed after the router was finished) would eliminate explicit submachine management, and give a much clearer picture of "what happens next" on a given connection.

7.3. Process Manager

The process manager handles host-specific process creation and destruction for the HPC kernel. There is a separate manager for each physical host. As with the IPC router, the manager-kernel interface consists of a TCP/IP connection and an asynchronous command protocol, but the process manager is a much simpler piece of software.

The kernel passes the arguments from an **animate** to the process manager for the specified host. Inside the manager, a lightweight task is created to fork and exec a process with the specified arguments, then return the host process identifier to the kernel. The process manager also monitors the processes it has created and reports termination to the kernel, which translates process death into implicit die operations.

By design, resources like files and devices are manipulated outside the HPC system. However, many operating systems protect such resources by using access control lists and associating some user identity with each process. The HPC process manager creates processes with an user identity without any special privileges, and the UNIX set-user-id mechanism can be used to associate additional privileges with a specific executable program. This interface to the host protection system is not strong or flexible enough to protect independent HPC applications from one another, but a better solution is a matter for re-design rather than implementation.

7.4. HPC Kernel

The kernel process is internally organized into several packages of software with sets of related lightweight tasks. Altogether, there are nine distinct packages built on tens of supporting libraries not specific to the kernel. The major packages can be roughly divided into those that handle interactions with the world outside the kernel process, and those that handle strictly internal functions.

The client package handles four classes of client-kernel interactions: registration, terminal setup, invocation of primitives, and asynchronous notifications. There is a permanent task to wait for new client connections, a transient task to handle the body of the registration protocol, and a task to handle synchronous client interactions.

The router and process manager packages resemble each other, with one permanent task waiting to demultiplex incoming messages, a small database of tasks waiting for messages with specific tags, and a permanent task waiting for unsolicited terminal or process death notifications. An RPC-stub module conceals the details of the protocols from other kernel tasks. The router package also contains stub-like routines for terminal setup that run the client-kernel and router-kernel subprotocols concurrently to minimize latency. (Such routines are obviously unable to use the RPC-like blocking interface provided for other client or router functions.)

At the heart of the kernel is a structural database with complex layered routines to access the database, and a task for each pending invocation. Client tasks spawn these operation tasks in response to explicit agent invocations, and the terminal and process death tasks spawn them in response to failures or uncooperative terminations.

This gives central responsibility to the client task, since most invocations are started there. This distorts the kernel's natural structure, because all HPC primitives are fundamentally applied by domain, rather than by agent client.

The final package is an internal controller service. There is a permanent task that waits for creation of new complex domains, and creates controller tasks to handle each one separately. Controller tasks monitor domains for permanent and temporary losses of control, and apply the appropriate policies. In a better implementation, each controller task would have an IPC terminal for receipt of control messages, and the controller tasks would spawn operation tasks instead of the client tasks.

Figure 7.1 shows the overall structure. Circles denote tasks, rectangles represent modules shared between tasks, and triangles show queues on which internal tasks may block. Solid lines show regular calling patterns and dashed lines show task spawning. Despite appearances, there are no circular dependencies between layers.

7.4.1. Database Operations

Each operation task calls a single entry in the network layer of the structural database software, which is responsible for translation between external protocol and internal data representations. A network function reads and decodes the body of a protocol message into convenient internal data structures, verifies that all purported HPC identifiers are legitimate, and passes on to the next layer. The network layer reports errors of all kinds back to the invoking agent.

The high-level semantics layer completes translation of arguments, does argument validation, and requests any necessary serialization. The function for a given HPC primitive first converts HPC unique identifiers into pointers to the appropriate database structures, then collects any structures affected by the operation that are not explicitly named in the argument list. The function then checks the domain of the request, the types and domains of the arguments, and the structural relations among the arguments against the preconditions of the operation.

The high-level layer brackets calls to the low-level semantics layer with calls on the synchronization layer to ensure conflicting operations do not overlap. An operation will be blocked until no possibly conflicting operations are active. When resumed, it must check its argument list for changes caused by such conflicts, possibly collecting a new set of implicit arguments. This synchronization is obviously needed in a decentralized or truly concurrent kernel, but it is also needed even in this logically centralized, non-preemptive tasking design. The interfaces to host IPC and process managers allow an operation task to block, yielding the kernel to another task while a remote operation completes

After checking and serializing, low-level semantic routines are called to do the significant manipulation of abstract structures. This level encodes HPC structural semantics, calling on a bottom-most database layer to record abstract structure, and the IPC and process manager packages to realize abstract changes in physical resources. The task handling the client registration protocol directly invokes some utility functions in this layer to create new shells and controllers. The low-level semantic layer also issues all structural change notifications and responses to inquires.

The HPC database does not directly implement all the relations like *adjacent* in the formal specification of HPC structure. What is elegant, or at least simple, in a formal setting may be absurd in a computational setting. For example, when incrementally modifying structure, it is much more efficient to infer adjacency from explicit spaces

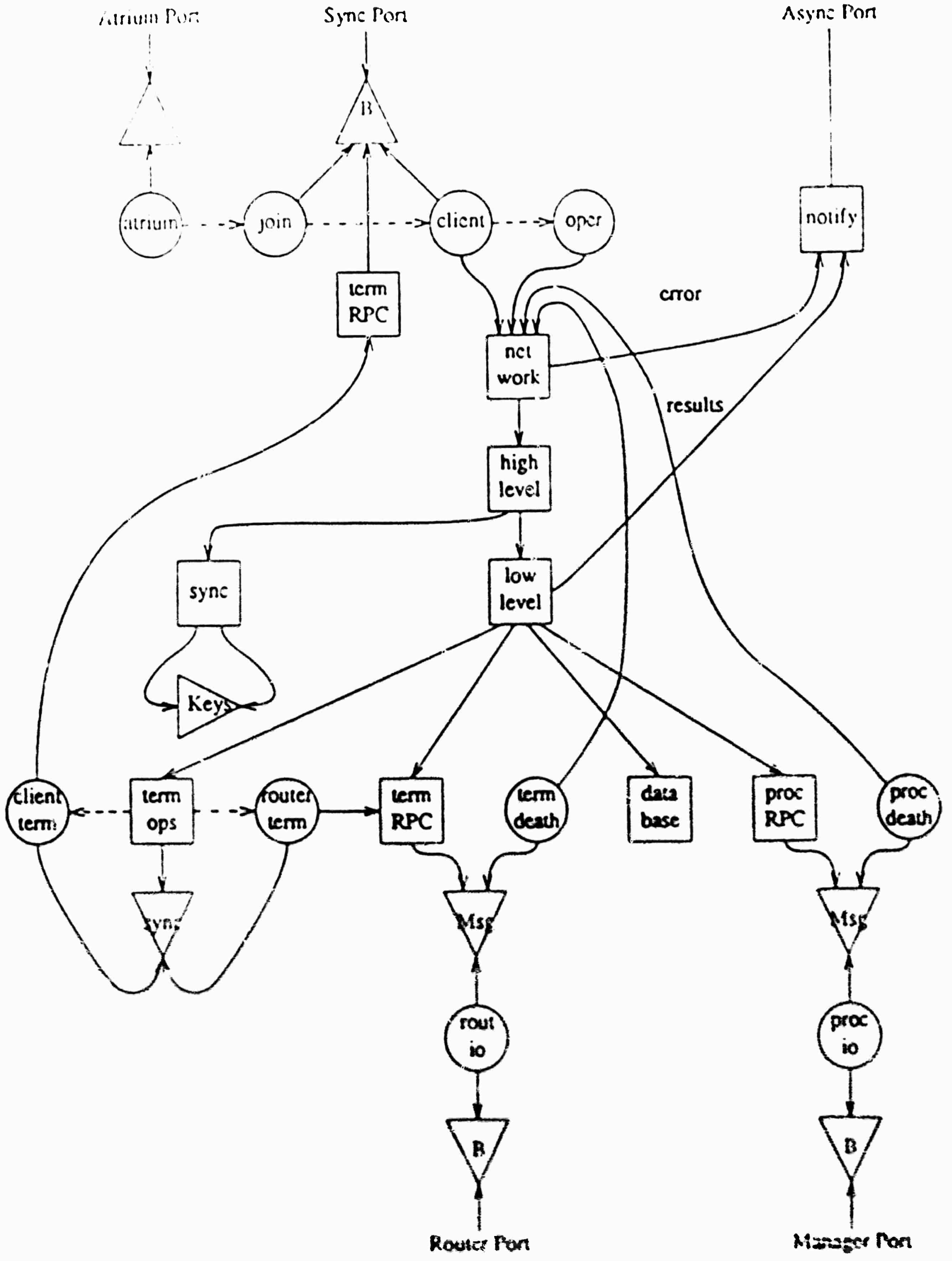

Figure 7.1. Kernel Software Structure

in the database than infer the spaces from an adjacency relation.

The most complex data structures and algorithms in the kernel are in the low-level semantic layer to handle path maintenance. These routines will be discussed in detail later.

The final database layer is the collection of basic data structures and their access routines. It provides a name registry for use by the network layer to translate external HPC identifiers into internal pointers. There are basic internal data types for spaces, views, shells, domains, and processes, though only shells, views, and domains can be named by clients.

Each type has operations for initializing its module, and for creating, validating, and freeing structures. Each type representation carries a unique password value in its first location, and the database and the low-level semantic layer routines rigorously check each argument for pointer and password validity. Besides these generic operations, each type has specialized operations to set and clear fields, the most important being pointers linking two structures to each other. These operations ensure that both links are made and broken at the same time, that links to structures are never lost by accidently overwriting them, and enforce a useful discipline in the low-level semantic routines. Figure 7.2 shows the most significant links between the internal data structures.

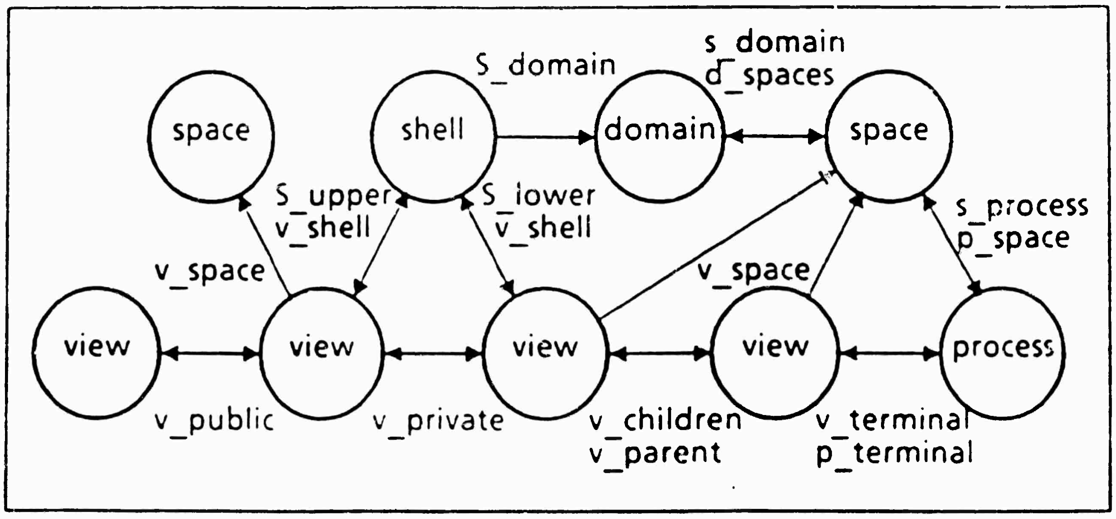

Figure 7.2. Principal Database Links

The database layer uses generic data structures like hash tables and ordered sets throughout for fast lookup and arbitrary numbers of similar links.

7.4.2. Root Domain and Controller Service

A special case in the second step of splicing allows promiscuous services. If the remote shell has been spliced to a well-known identifier, then a sibling of it is created and spliced to the local shell. This single well-known identifier amounts to a *service service*, because shells spliced to that identifier may then be spliced to from clients at arbitrary places in the hierarchy without prior negotiation, once their identifiers have been distributed. It was more convenient to provide services this way than implement the restricted number of service shells described earlier.

The distinguished root domain is treated only slightly differently from client domains. The low-level semantic layer delivers structural change notifications internally to a permanent "root agent" lightweight task instead of externally to another process. The root agent task has two functions. It cleans up top-level domains that abdicate by killing their subtrees, and it uses the service service mechanism to implement the internal controller service.

During start-up, a shell is created inside the root domain and spliced to the service service. When processing invest, the kernel splices the controller shell in the new complex domain to this controller service shell. The root agent task creates a separate controller task to monitor each new domain.

The controller service is a fairly typical service, despite its implementation inside the kernel. A task watches over the overall state of the service, clients come and go, tasks are created to service them, and (in a proper implementation) client requests are translated into operations on an internal database. The only reason it couldn't be reimplemented outside the kernel (cf Section 3.4) is that liveness on the controller interface is insufficient to distinguish permanent and temporary losses of control.

7.4.3. Path Maintenance

Both the client interface and the formal specification deal directly with connections, shells, and splices. These define local connectivity between views. Global, end-to-end, connectivity between objects is a complex function of the local connectivity, involving indirect bindings introduced by abstraction (corresponding components) and by composition (chains of alternating public and private peers).

Deducing global connectivity from the local connectivity is certainly possible, and the formal specification is written that way for simplicity. However, obtaining deductive closure directly from axioms of direct binding is unsuitable for any real system. The efficient, incremental computation of global connectivity and liveness triggered by operations on direct bindings is the most interesting algorithm in the kernel, called *path maintenance*. (The similarity to truth maintenance systems in implementations of formal logic is deliberate.)

Path maintenance keeps track of direct and indirect *bindings* between views. The indirect bindings retain enough global information to compute the effects of a change quickly, regardless of the structural distance between the cause and the effect.

Each view data structure maintains separate lists of private and public peer bindings. Private and public bindings are distinct, and a pair of views may be bound both ways simultaneously. Only direct public peers (connections) are used in path maintenance. However, a view's list of private peers includes views at all odd distances down chains. These cached bindings allow propagation of changes in connectivity and liveness directly to distant affected views.

Multicasting allows more than one chain between two views, creating a given indirect binding in more than one way. Path maintenance retains the proximate *justifications* for each binding to ensure they are removed at the correct time. Direct bindings have a primitive justification as a connection or a shell/splice. Indirect bindings between corresponding components are justified by the binding between their parents. Indirect bindings between peers along a chain are justified by the alternating private and public bindings in the chain.

Bindings and justifications form a directed graph where the sources are the primitive justifications. At odd distances from the sources, the entries are bindings; at even distances, justifications.

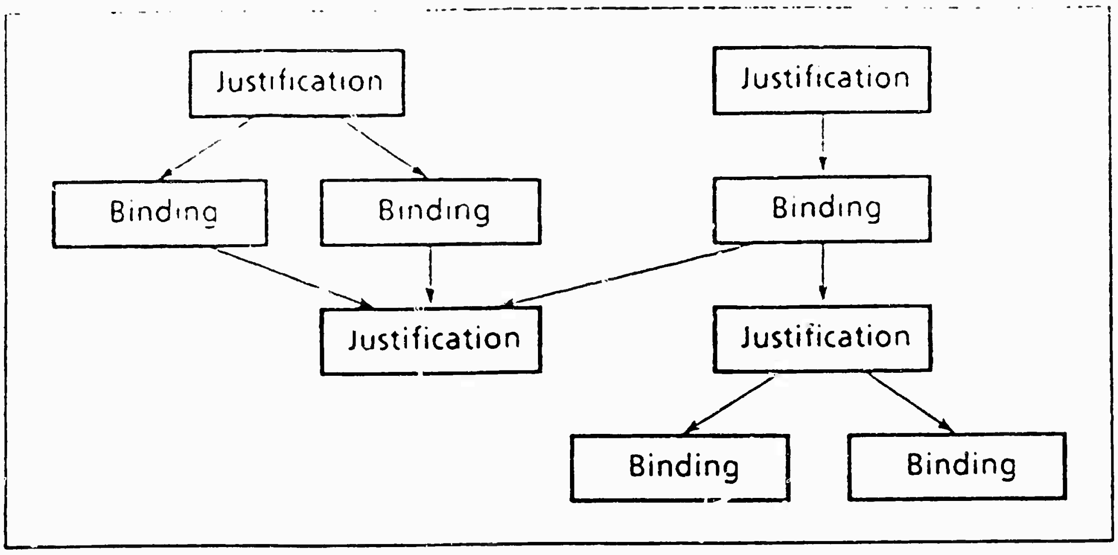

Figure 7.3. Binding-Justification Graph

Inferring the effects of a local change from the direct bindings requires the minimum amount of space, but an unreasonable amount of time. The corresponding binding/justification graph would be trivial, with just primitive justifications and direct bindings, and every view along a chain must be visited on every related operation. Recording the complete set of direct bindings that ultimately justify an indirect binding represents the other extreme, because each effect can be looked up in constant time using a binding/justification graph four layers deep: primitive justifications, direct bindings, derived justifications, and indirect bindings.

However, direct lookup is expensive in space and has substantial hidden costs in time. The cost of creating or destroying a single justification grows with the distance between the bound views, because links to a greater number of justifying bindings must be maintained.

Path maintenance compromises between lookup and inference to improve performance. It deepens the binding/justification graph by allowing indirect bindings to justify others, while reducing the fan-in and fan-out of individual bindings and justifications. Because the graph is no longer bounded in depth, inferring an effect of changing a direct binding is no longer a constant time operation. But neither is it necessary to visit nodes that are unaffected by the change.

Bindings between corresponding components are justified by the binding between their immediate parents. Bindings along a chain are justified by just three bindings. For a binding between views v1 and v2, one direct connection between, say, c1 and c2, and the private bindings (direct or indirect) between v1 and c1 and between v2 and c2 are recorded. This justification is not unique, because any connection between v1 and v2 could be chosen. The order in which bindings are added to a chain determines the specific trio used to justify a binding.

This compromise has several notable points.

- The apparently greater cost of traversing the graph (inferring indirect effects) is actually of the same order as the cost of traversing a complete list of precomputed effects. Both traversals reach the same views and bindings.

- There is a large reduction in storage requirements for links between bindings and justifications. The same number of bindings exist, but every justification now has a bounded number of constituent bindings, and bindings for views closer to the tops of their local hierarchies participate in many fewer justifications. (In exchange for this global reduction in links, cycles may introduce a larger number of justification data structures.) The reduction in fan-in and fan-out translates into increased speed when manipulating justifications.

- The binding/justification graph is not a DAG. Cycles allow paths between peers with and without complete pass around the cycle. A binding that does not depend on a cycle can justify itself when a cycle is completed by adding a connection. Figure 7.5 illustrates a cycle in a part of the binding/justification graph resulting from the cyclic path shown in Figure 7.4. Longer path cycles can create longer cycles in the binding/justification graph.

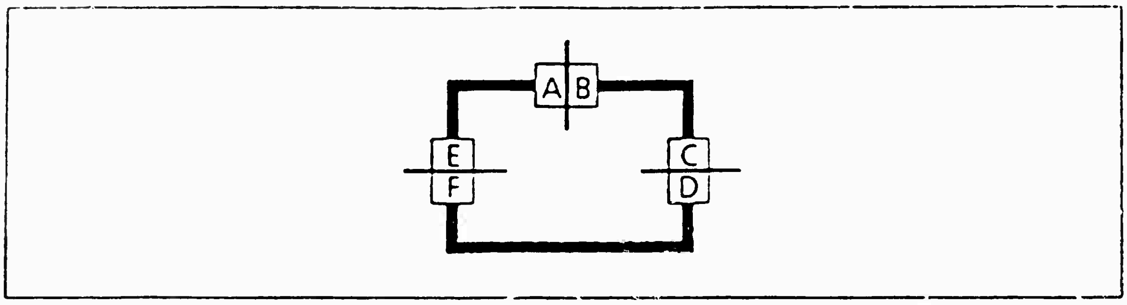

Figure 7.4. Typical Cyclic Path

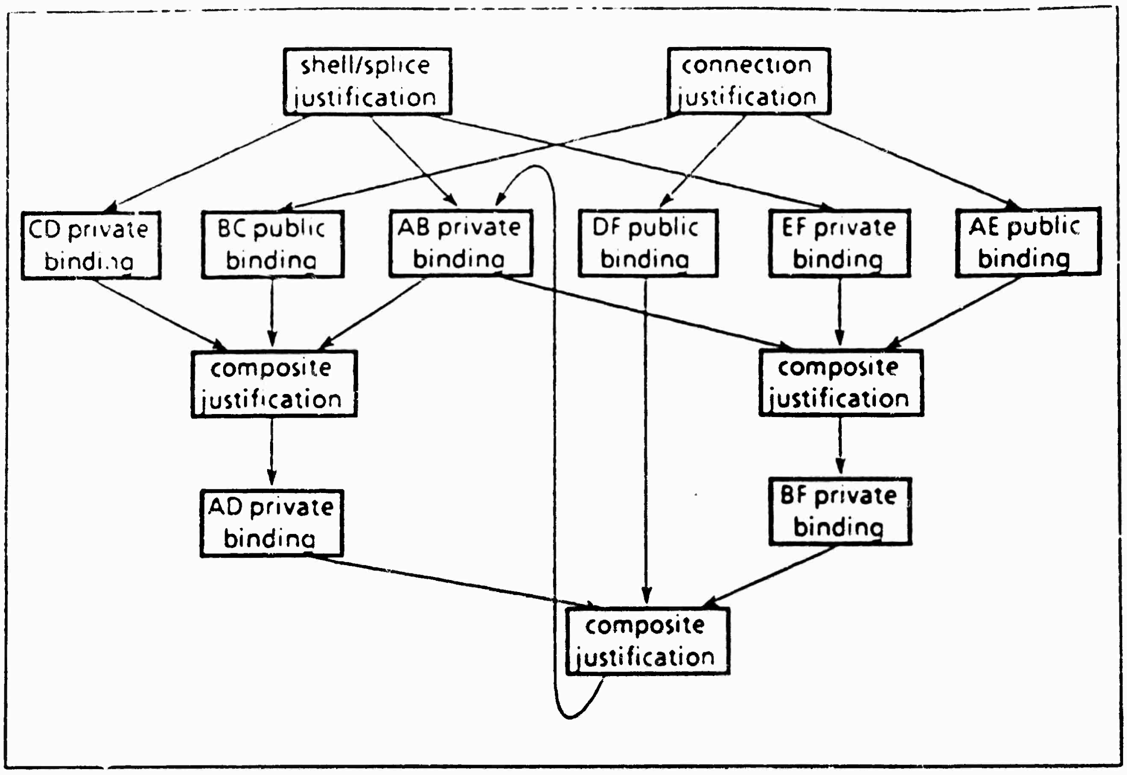

Figure 7.5. A Resulting Self-Justifying Binding

The kernel translates connect and disconnect into creation and destruction of public bindings. Similarly, client shell and splice manipulations are translated into creation and destruction of private bindings, in addition to manipulations on spaces and view hierarchies. These bindings are given a primitive justification. The *new* and *delete* operations affect structure within a view hierarchy, and the bindings between corresponding components are indirectly justified by their immediate parents.

Binding and justification removal follow a simple rule. A binding is removed when its last justification is removed, but a justification is removed when any of its constituent bindings is removed. This leads to a recursive algorithm for destroying connections or shells/splices. Addition of bindings and justifications naturally obeys the converse rule, but is more complex because it must compute any additional bindings justified by a new binding, while the destruction algorithm simply looks them up in the binding/justification graph. Pseudocode for these algorithms is shown in Figures 7.6 and 7.7.

In Section 4.2 we noted that multiple paths between views should not lead to multiple delivery of messages. Any pair of views has just one (private and public) binding in the database, regardless of the number of justifications, so the IPC router is easily instructed to provide single delivery. When a binding is justified a hash table is searched to determine quickly if it already exists.

A greater potential problem is the infinite number of chains between views provided by zero or more passes around a cycle. An algorithm based on propagation (or inference) along a chain must include an explicit check for cycling. Path maintenance avoids this by adding bindings based on information at a fixed number of nodes, and adding justifications only when a new path is created. When a cycle is completed, a single justification accounts for all numbers of passes through it.

Ordinarily, cycles in the binding/justification graph would never be removed, because the underlying strategy is reference counting. However, all bindings are removed correctly without expensive checks for cycles. First, path maintenance prohibits justifications in which a binding directly justifies itself. This prohibition requires checking a binding against at most three others each time a new justification is found.

If direct private bindings could be removed at arbitrary times, this would be insufficient. However, direct public bindings have only primitive justifications and semantic constraints at higher levels ensure a direct private binding will never be destroyed while it is part of a cycle. Shells, and the direct private bindings between the tops of their interface hierarchies, are destroyed only by disclose and splice. A precondition for disclose is that there are no connections to any view on either side of the shell, and a precondition for splice is that there are no connections to any view on the lower side of the shell. Induction can prove that the loops in the binding/justification graph are removed before it is legal to destroy a binding that indirectly justifies itself.

Because a unique path between two views is justified only once, the non-unique three-binding path representations are permissible. In direct lookup, mentioned above, each derived justification uniquely represents a non-cyclic chain. In the three-binding compromise, each justification still represents a single chain, but a chain will have multiple representations. These multiple representations account for the possible growth in the number of justification data structures. However, different representations are used to justify distinct bindings, never to justify a single binding redundantly, and never for bindings that don't complete a cycle. Moreover, it is never necessary to search for a path, only for bindings.

The number of bindings grows as the square of the length of a chain, because path maintenance records indirect bindings between every pair of private peers. This is a direct and expected consequence of maintaining an explicit database instead of performing an exponential number of inferences. Creating of a single binding takes constant time, but recursively creates N additional bindings, where N is the length of the chain just created. For some applications, this linear cost would be unattractive, but HPC must visit all the views along the new chain to report changes in liveness.

The primary reason for bindings between all pairs of views along a chain is to allow destruction of bindings in the middle of the chain without explicitly traversing it. It is simple to bind only the ends of a chain together, and grow chains at the ends, with constant time creation and sub-linear (even negative) growth of maintenance data structures. However, destroying a connection requires identifying the ends of all chains the connection justifies, either directly or indirectly through corresponding components. An alternate path maintenance algorithm based on direct bindings and maximal chains is worth investigation, but proper handling of cycles, reflectors, and multiple paths may reduce its apparent advantages. Anyway, graph size has not been a problem in practice, and HPC requires visitation of all nodes for other purposes.

```
binding_create(v1, v2, type, j)
view v1, v2;
int type;
justification j;
{
  create b = new binding(v1, v2)

  add j to b's justifications

  if both views are terminals
    create a transport connection

  // add any bindings justified by this one
  if type is PRIVATE

    for all public peers p of v1        // collapse to right
      for all private peers q of p
        add_binding(v2, q, [b, v1-p, p-q] )

    for all public peers p of v2        // collapse to left
      for all private peers q of p
        add_binding(q, v1, [q-p, p-v1, b] )

    for all children p of v1            // collapse corresponding components
      for all children q of v2
        if p and q correspond
          add_binding(p, q, [b] )
  else

    for all private peers p of v1       // collapse connection
      for all private peers q of v2
        add_binding(p, q, [p-v1, b, v2-q] )
}

add_binding(v1, v2, constituents)
{
  find or create j = justification(constituents)

  find b = binding(v1, v2)

  if b must be created                  // no existing justifications
    binding_create(v1, v2, PRIVATE, j)
  else if b is not in constituents      // no direct self-justification
    add j to b's justifications
}
```

Figure 7.6. Binding Addition

```
binding_destroy(b, j)
binding b;
justification j;
{
  remove j from b's justifications

  for all justifications j' with constituent b
    for all bindings b' justified by j'

      if b' must be destroyed           // loss of all justifications
        binding_destroy(b', j')
      else
        remove j' from b' 's justifications

    destroy j'                          // loss of any constituent binding

  if both views are terminals
    destroy network connection

  destroy b
}
```

Figure 7.7. Binding Removal

7.5. Tools

While implementing the HPC prototype, we enjoyed, suffered through, or craved a variety of tools and programming techniques. Time spent in building basic tools or "wasted" in disciplined program design and testing will be more than repaid in reduced overall development time and reduced maintenance. Sometimes tools and tool building make the difference between a success and a completely unmaintainable write-off. This is as true of experimental software subject to frequent and rapid change, such as HPC, as it is of commercial codes.

HPC code quality is high. Most packages are very robust and easily modified. Some of the tools used to keep them that way, along with some of the failures, are worth reporting.

7.5.1. State Table vs Tasks

As mentioned earlier, two substantially different methods of writing multiplexed programs were used: a table of state machines and a collection of non-preemptive, lightweight tasks. A state table can be implemented using the simplest of tools, and it is an intuitive approach. Those are its only merits, and a small investment in tools offers a large reward.

The heart of the tasking package is a simple coroutine package, but HPC never uses the coroutines directly. Above the coroutines are queues, a round-robin scheduler task, and routines for tasks to sleep on a queue, wake up a queue, yield control to the scheduler, and terminate. The scheduler task runs a "backstop" task to collect external events (like host I/O) when all other tasks are sleeping. These tasking functions are convenient and unintrusive to use.

The tasking package has two big advantages over state tables. The backstop task distributes external events without knowing what to do with them, and regular tasks just wait for the events they want without dealing with the distribution. This separation makes both distribution and processing of events cleaner, easier to read, and easier to extend. This is more important that it might seem at first. The HPC kernel redistributes many external events two or even three times. For example, the backstop task waits on the host for external events, the process manager input task waits on the backstop task for messages from the manager, and the process death task waits on the manager input task for death messages. Tasks cleanly separate the responsibilities and concerns of the three levels.

Second, tasks encode significant flow of control in one place using a conventional programming language. In the state table approach, flow of control is encoded in the data, and distributed over many different functions. It is hard to distinguish the logically separate computations, and hard to manage interactions between them in a state table, but easily managed using queues for synchronization with associated data structures for communication. Development, maintenance, and debugging are all much more difficult for state tables than tasks.

Lack of compiler support is the only notable disadvantage to lightweight tasks. Non-preemption has been an advantage, making every block of code an implicit critical section and eliminating the need for nuisance locks on data strucaures, rather than a disadvantage. The underlying coroutine package requires architecture- and compiler-dependent assembly coding, however, the original package was easily ported to diverse architectures such as VAX, MC68000, and ROMP (IBM RT-PC).

The real problem is stack management. A task's maximum stack depth is fixed when the task is created. Stack overflow was a continuing problem during development. Overflow can not be automatically detected without compiler support, so the task package was extended with a mechanism to measure the deepest point reached on a stack. The main function for each kind of task is coded to measure stack depth explicitly after each iteration through its body and, unfortunately, after any damage has been done. Packages are recompiled to increase the stack size associated with tasks that come too close to their limits. Many tasks have bounded stack usage, but operation tasks call recursive routines, so any limit may be insufficient.

7.5.2. Message Library

The synchronous, master-slave, paradigm offered by RPC is inadequate implementing for the generally asynchronous, highly multiplexed, peer relationships between the component processes of the HPC system. Therefore we built RPC-stubs that block tasks instead of processes and allow the flexible distribution of incoming messages to existing tasks needed inside the HPC kernel. Compiler and stub generator support would have been welcome, but not with the additional baggage carried by available RPC implementations. In particular, network transport, message encoding and decoding, and flow of control had to be managed separately in the HPC kernel, while these are all unified in RPC.

The ideal transport medium was reliable (perhaps unordered) delivery of messages with distinct boundaries. Unreliable delivery of messages (UDP) and reliable delivery of byte streams without internal boundaries (TCP) were available when implementation started. We chose to build rigidly formatted messages on top of TCP, pray for detection of malformed or unsynchronized messages, and resynchronize after errors by dropping the TCP connection. This was considered a better investment than implementing our own reliable transmission protocol.

There is no question that the HPC implementation would have died a miserable and lingering death if application messages were assembled and encoded "manually". To centralize byte packing and conversions between internal and external representations, a message library was written that would take a message buffer and a human-readable format string and scatter or gather the arguments specified by the format. Format strings use abstract data types relevant to HPC (shell) rather than the underlying concrete types (long).

The self-documenting format feature was a great success, explicit management of message buffers was tolerable when packaged correctly, but the external data representation was a serious mistake. The external representation used one, two, and four byte quantities *aligned on quantity boundaries* from the beginning of the message, under the impression that this would avoid problems caused by host data alignment requirements. What actually happened was that entire *messages* had to be aligned to be read properly, and that alignment had to be maintained even after the first part of a message had been discarded.

Eventually a solution to this alignment problem was neatly, even elegantly, packaged up, but fixed size data quantities would have been a better solution. A still better solution would have been to use an existing de facto standard encoding like XDR or Courier, preserving the message library to translate between HPC abstract types and the concrete types directly supported by the encoding.

7.5.3. Protocol Grammar

The message library deals with individual messages, but does not simplify programming exchanges of multiple messages or of demultiplexing interleaved exchanges. These are problems both for the kernel, and for any realistically complex agent, which must manage several concurrent but independent strategies for different portions of its domain. The client interface provided by HPC must be augmented with a wide range of programming support tools before the overall system is practical.

The user interface dialogue grammars under development by Yap [ScY88] look like attractive tools for managing many of these protocol problems when building an agent. We have not yet applied them in the HPC system.

7.5.4. Test Scaffolding

No exact records were kept, but it is probable that the disposable test scaffolding used for HPC development is larger than the HPC code itself. At least one, and usually two, test processes were created to take the place of clients, kernels, and routers. Each application protocol between processes with its corresponding interface libraries was tested independently of the application code, and IPC terminal setup was tested with nearly every combination of dummy clients, routers, and kernels during various stages of development. The earlier test processes were run interactively to step through each protocol and set up stress cases. Later scaffolding generally ran automatic test sequences to verify that further development had not introduced new bugs.

As a result of protocol testing, the kernel packages that handle interactions with external processes were debugged independently of the central database operations. The layers of database routines lent themselves to easy testing from the structural database up. As functions were added to each layer, corresponding test functions were added to a test suite and run after every significant change. The time spent running the test suite paid for itself in development time through early detection and good isolation of errors.

In many cases, fully exercising a layer in isolation violates global structural constraints. As mentioned earlier, the path maintenance code may not remove direct private bindings at arbitrary times. Isolated test exercises had to be gradually removed from the test suite as the dependencies between layers and checks for additional HPC constraints were added. The final kernel test suite contains over 650 calls on the low-level semantic and database layers, and over 950 checks on the results in the structural database. This is larger than any package except the low-level semantic layer, and none of it is used in the kernel.

7.5.5. Log Files

Log files are an invaluable, if unexciting, diagnostic tool. HPC keeps logs with adjustable levels of reporting for the kernel, IPC router and process manager. The globally unique number generation system also uses a per-host special log to prevent reuse of numbers.

A standard log message heading with the date, time, and logging process identifier was especially helpful in sorting out the variety of messages in the kernel and router logs. At various times during development, the log have recorded debugging traces, task stack depths, rough timing estimates, client arrivals and departures, internal task creations, dumps of valid incoming messages, dumps of protocol violations, resource consumption reports, hash table statistics, rates of unique number generation, host fatal errors, and violated internal assertions. Mundane, but essential, material.

7.5.6. Graphic Interface

Complicated dynamic activities can be very difficult to understand. Used wisely, graphic displays and user interfaces can make an impossible task practical, especially for a system like HPC with a natural graphic representation. An interactive graphic interface for domain agents would be a tremendous experimental tool. Unfortunately, good graphic interfaces are a major investment in development time and resources, and client support tools were not a important issue in this thesis.

Still, a simple, non-interactive graphic display was integrated into the tasking package to display the internal status of the kernel conveniently. Each task has a separate marker with descriptive labels. The display places the markers for different types of tasks (client, operation, controller, etc.) in different columns. This has been a valuable tool. The degree of multiplexing and the number of clients is manifest. Premature task termination, failures to terminate, and unreclaimed resources are instantly and obviously visible on the display. Log files provide the same raw data in a format that is *much* harder to use.

7.5.7. Interface Preprocessor

Interface structure descriptions (medium, orientation, component structures, etc.) are complex pieces of information that must be manipulated efficiently by software, transmitted via network connections, and communicated to and from human beings. There are three corresponding representations: linked graph data structures, linear encoded byte sequences, and ASCII strings.

For many purposes, a structure descriptions is converted from one representation to another at runtime. However, this is inconvenient for programming clients. A programmer would like to write an ASCII description inline in a program, and have it converted to a useful form during compilation. A simple source code preprocessor takes inline HPC descriptions and converts them into C language arrays initialized to the linear encoding for the description. This form can be passed directly to the client-kernel interface library, which is the most common use of inline descriptions.

7.5.8. Code Assertions

The low-level semantics and structural database packages must maintain many invariant properties to preserve HPC properties and avoid corruption of data structures. Assertions about these invariants are a significant fraction of the executable code in these layers. Seven percent of the low-level semantics layer, and 12 percent of the database layer is devoted to checks that are usually unnecessary and often redundant.

However, the cost of invariant checking is insignificant compared to the development and maintenance time it saves, not to mention the increased confidence it inspires in complicated database manipulations. On dozens of occasions, an apparently innocuous code addition or change violated an assertion. Usually, the violations were the result of incorrect coding that was acceptable i[illegible] context of the module being added, but incorrect in the larger context of the entire database. Asserted inv[illegible] were almost never too restrictive; the exceptions were due to either a lack of forethought in specifying t[illegible]nt o a radical change in implementation requiring global changes throughout the database.

Chapter 8 Conclusions

8. Conclusions

We began with three general goals: develop a structural representation for target applications, provide operations to manipulate the representation during execution, and identify specific influences of the distributed environment on application structure and management. As in any research, the successful pursuit of initial goals leads to unexpected conclusions and suggests goals of future research. Here we present some general conclusions on system design. We also suggest several research areas ripe for additional work.

8.1. General Observations

During this research, we came to some conclusions on system design that apply widely.

8.1.1. Dynamic Structure

- Use *semantics*, not *syntax*, to describe dynamic structure.

The typical language-based approach to distributed programming handles static process structures well, while handling open systems and run-time reconfiguration poorly, if at all. Dynamically changing structure should be represented as an abstract data structure with a set of manipulating operations, not as a syntactic form in a program.

The HPC design successfully demonstrates the data structure approach, encoding a broader set of process structures than any programming language we know. HPC also tackles dynamic changes and merge inconsistencies, which can not even be *expressed* syntactically. Future distributed programming languages should not attempt to encode process structure syntactically, unless the process structure is to be entirely fixed.

8.1.2. Process Structure

- Passive hierarchies are an appropriate model for overall application structure.

Many distributed applications can be naturally described as a nested hierarchy of abstractions. We find passive hierarchies superior to active ones on the basis of clean abstraction and fault tolerance through redundancy. However, passive hierarchies do not encompass all useful application structures. Most notably:

- A comprehensive model of process structure *requires* non-hierarchical features.

A strict hierarchy with explicit composition is too cluttered, too concrete, and too brittle to support complex applications in an open environment. While programmers and managers may be presented with the appearance of a strict hierarchy, practical systems require controlled violations of this paradigm to provide transparency. Also, trees are intrinsically incapable of expressing the merge inconsistencies that they can generate. It is far better to use arbitrary graphs as a basic structural model, and impose a hierarchy as a surface feature, than to use trees as a fundamental model. This conclusion was not expected, but our hierarchically motivated design was not complete and consistent until we adopted graphs as the underlying model.

8.1.3. Management Structure

- The relation *A manages B* is as important as *A communicates with B*.

Powerful tools are needed to describe and dynamically manipulate this relationship between agents and domains. In the context of HPC, this observation suggested the reuse of existing compositional tools. By adding abstract placeholders for each domain, the complete protection relation can be described in the same explicit detail as the communication relation.

Before reusing composition, we investigated a number of inheritance and default rules for propagating privilege from an abstract, multiprocess object to (some of) its real processes at the leaves. All rules led to conflicts with the basic principles of abstraction and composition. In contrast, the division of the hierarchy into domains that follow object boundaries was an obvious, and satisfactory, design decision.

This observation applies widely. A great many tools, ranging from network protocols, through operating systems and programming languages, give close control over binding two or more communicating entities together. In contrast, the tools for specifying protection and management relationships are crude and limited in most environments. In designing any new software system with dynamically changing structure, the same degree of care should be given to a powerful set of tools for relating agents with domains as to the rest of the system design.

8.1.4. Communication Structure

Communication functions can be classified as logical configuration, physical implementation, and end-to-end communication.

- Each class of communication function should be provided independently.

Workers communicate, managers configure, and the HPC kernel implements. This separation is not provided by current IPC mechanisms and operating systems. One specific consequence of this classification is that:

- Efficient separation of implementation from communication requires a *third-party connect*.

We can not actively manage a distributed computation without making changes to it, and existing network protocols do not allow an HPC kernel, for example, to set up or tear down connections between workers in an efficient way. Third-party connect is an important session layer feature that should be incorporated in future protocol suites.

- Complex communication patterns can be expressed structurally.

In contrast to implementation (third-party connect), it is *not* necessary to support configuration in the communication protocol. The paradigm of point-to-point connections between interfaces does *not* limit a system to one-to-one communication patterns. It is not necessary to rely on details of addressing or routing to manipulate heterogeneous parallel channels, homogeneous multiplexing, or multicasting.

- On-line computation of connectivity is practical, but non-trivial.

The most interesting algorithm in the HPC implementation is the incremental computation of communicating peers, in which the HPC kernel converts configuration information into implementation decisions. Our best algorithm is a centralized one that computes the effects of connect and disconnect in time proportional to the *length* of all the affected paths. We unsuccessfully sought an algorithm with cost proportional to the *number* of affected paths (effectively constant time). We would also prefer an decentralized algorithm that could be distributed.

The HPC path maintenance algorithm is similar to on-line algorithms for transitive closure, a formal property with many practical applications. Therefore, we expect path maintenance to find use outside the context of HPC. Path maintenance also has many potential applications in distributed network routing algorithms.

8.1.5. Distribution

- Distribution mandates an asynchronous system interface.

This point is controversial, but our experience is that synchronous interfaces are appropriate only for systems with centralized behavior. To capture the essence of distribution, one must allow for the intrinsic asynchrony of multiprocess programs, and of failures. Transactions (and atomic non-primitive actions) are not consistent with highly-available access to a distributed data structure, because multiple agents may be inspecting and modifying a shared data structure concurrently. Asynchronous notifications of change are needed to avoid expensive polling, just as interrupts are needed to support efficient operating systems. Support for multithreaded agents also requires an asynchronous interface, to allow overlapping operations issued by several threads of a single agent.

Building synchronous interfaces for programmer convenience on top of the basic, asynchronous, system interface is compatible with a distributed environment. For example, a programming environment might use several lightweight processes to wait, synchronously, for each of several overlapping operations to complete. But such programming environments are successful by virtue of what they hide. Eventually someone will need access to the ugly asynchronous reality, even if only to build a better environment.

- Partitions do *not* require reduced availability.

Put another way, if you know what you're doing, you can do it more often. Most databases understand nothing about the semantics of the data they contain, and therefore cannot resolve merge inconsistencies in a sensible way. As a result, database designers strictly avoid consistency problems by limiting availability. The more that is known about the application, the more restrictive this strategy becomes.

The Locus distributed file system is a specialized database that knows the semantics of much of its data, and exploits that knowledge to reconcile automatically many inconsistencies at merge time. HPC maintains a richer, even more specialized, database, and understands almost everything about its data (at the expense of containing only specialized data). Merging is based solely on current partition state, using a history-less algorithm, and most structural features can be reconciled automatically.

Our experience suggests that application complexity is not a basic obstacle to availability during partitioned operation. In fact, we offer this heuristic to generalize the comments just made:

- The greater the number of internal constraints a specification has, the fewer the external constraints an implementation will have to add to operate in a failure-prone, partitionable environment.

8.2. Suggestions for Future Research

Our experiences with HPC suggest several areas for investigation, including specific improvements to HPC, general network services, and semantic models for concurrent programs.

8.2.1. Design Extensions

It is usually hazardous to allow more features to creep into a satisfactory system. However, there are at least two areas of the HPC design where additional features deserve investigation.

- Allow user-specified correspondence of views.

HPC's fixed definition of corresponding views leads to the tap problem, the user inability to detect cycles, and related problems. It may be possible to find one mechanism that provides solutions to this whole set of problems. For example, users could assign endpoints tags or markers that would propagate along paths and define corresponding and reachable views. Integrating such a scheme into the design of complex communication paths (e.g., multicasting), and the implementation of path maintenance is the technical challenge.

- Place limits on splice targets.

As previously noted, hidden communication paths are an improvement over strict hierarchies, but they are not always a good thing. It is easy to express limits on the targets of splices, for example: "must be within subtree T". But it is not obvious how to check such constraints quickly, nor how to integrate them into the existing protection system. In the current HPC design, no domain can limit what another domain does internally. One domain can impose its will on an adjacent, inferior, domain only by taking full control over the inferior domain.

8.2.2. Related High-Level Services

HPC provides a structuring service. Alone, it is not sufficient to build complex, distributed applications. Many of the other necessary services (transport protocols, file service, name service, remote execution) already exist in most environments, but we found the need for some network services not yet available.

- Dynamic property (arbitrary string) service

HPC currently maintains two uninterpreted properties: role and type names. This was pragmatically the right thing to do, but wrong in principle. Users should be able to attach arbitrary properties to structural items, as long as proper operation of HPC does not depend on them, and the HPC kernel need not be extended to support them. Instead, properties should be stored in a name service allowing dynamic registration of arbitrary data. The DARPA domain name server demonstrates the technology needed to support name lookup for a restricted class of properties changing fairly slowly. The proposed X.500 directory service has optional support for unrestricted properties, but vigorous development is needed in this area, especially to allow users to quickly and automatically establish naming sub-domains.

- Network-wide credentials and authentication

HPC uses only TCP sequence numbers and IP source addresses to authenticate communication between components of the system. This is very weak protection against a spoofing attack on the HPC implementation. Additionally, HPC has no notion of user identity, and cannot provide its host sites with information needed for access control and resource accounting purposes. This brings access to resources down to the lowest level: anonymous guest.

Traditional resource sharing on the Internet is accomplished by creating local user accounts *within* an administrative boundary (e.g., MIT Multics), and using local authentication (login during telnet).[11] Using password challenges when programs, rather than people, must be authenticated is a bad idea, because programs can be examined for password strings. At a minimum, a network-wide credential and authentication scheme is needed before any significant automated resource sharing can be done *across* administrative boundaries. The Kerberos

[11] The original Arpanet visions of distributed resource sharing have never really been fulfilled. With few exceptions, the networking communities have stopped short at electronic mail, file transfer, and remote login.

authentication system used in Project Athena is a good starting point for further work.

- Session layer support for configuration

The third-party connect facility we found so important is a basic configuration feature that belongs to the session layer. We expect dynamic reconfiguration of distributed applications to require a range of session layer features beyond third-party connect, just as the current ISO proposals for session layer synchronization extend far beyond the original concept of data quarantine. Current work into automatic network management should be broadened to consider the necessary protocol support for automatic application management.

- Software development tools

HPC provides a raw, low-level environment, as expected of a set of basic mechanisms. Tools that incorporate some policies and allow programming at a higher level are needed, even at the expense of generality. The most glaring example is the lack of a standard interactive utility for HPC analogous to the many shell programs for the Unix operating system. The development of automated agents, perhaps customized for particular applications, is a more challenging research area that brings theoretical studies of distributed algorithms together with systems engineering and implementations. Another interesting problem area is the integration of HPC mechanisms with conventional fault tolerance mechanisms (transactions, redundancy, recovery).

8.2.3. Semantics and Formal Directions

HPC's need to manage a strict hierarchy with an undirected graph model suggests some extensions to formal semantic models, as well as the pragmatic tools discussed above.

- Remove the parent-child asymmetry in formal studies of semantics.

Many formal studies of the semantics of concurrency are based on passive process hierarchies as in CCS and CSP. The axioms of composition in such systems describe the behavior of a complex node as a function of the behavior of its children. However, the parent-child relationship is partly a matter of perspective. Any node can be chosen as the root of a CCS or CSP tree without affecting its behavior. The tree has exactly the same leaves composed in precisely equivalent ways, no matter which node is selected as the root, and therefore must have the same behavior. The sets of equations describing the various orderings of a tree often appear quite different. The laws of distribution for a formal system must be sufficient to prove that all such sets are exactly equivalent.

- Investigate semantics of nonhierarchical structures.

CSP and CCS systems cannot express sharing or transparent abstraction. There is only one path of interaction between two processes and all interactions between them are visible all along the path. There seem to be two technical obstacles to providing passive graphs with a formal semantics. The first is infinite families of equations, or of solutions to equations, due to cycles in the graph. The second is the loss of strictly local composition.

Our experience with HPC shows that families of formal solutions can be detected and reduced to a single representative, or discarded if no concrete solution exists. We conjecture that this experience can be extended from equations of connectivity to equations of behavior. The loss of local composition is more apparent than real. Every direct interaction between two nodes is explicitly represented by an edge. Solving the equations for cycles automatically handles any indirect interactions.

8.2.4. Distribution and Decentralization

A production quality HPC system would require significant improvements on the prototype implementation we constructed. Most of the work must go into application managers, and the HPC kernel also needs some revision. However, major improvements are beyond our current understanding of decentralized control, and we propose some research needed to support the development of distributed agents.

- Decentralize the HPC kernel.

The HPC interface to distributed applications and managers is satisfactory at this point, but the HPC implementation is too centralized. The implementation is built from several distributed processes, but the current HPC kernel can be neither replicated nor decentralized.

A preliminary investigation of a decentralized kernel indicated that many kernel functions could be readily distributed. The obvious way to divide the physical database is along logical domain boundaries, replicating copies of a domain only on hosts with a physical agent for the domain. The key problem area is decentralization of the algorithms, rather than data. In particular, it is not obvious how to distribute the path maintenance algorithm without communicating the entire path maintenance graph.

- Investigate decentralized control.

This thesis explores a set of mechanisms, without presenting policies for their use. We have assumed polices are determined by agents' behavior, outside the scope of our study. Behind this assumption is a challenging research area.

Any robust application must have multiple, distributed managers. Those managers must collectively agree on policy, and must further agree which particular manager is responsible for executing policy. Various special aspects of decentralized control have been studied in different fields: distributed agreement in the areas of theoretical distributed computing and reliable systems engineering, distributed control in the fields of industrial engineering and applied mathematics, distributing an invariant in theoretical distributed computing, flow and congestion control in protocol development and system modelling, and so on.

These fields of intense, specialized research can all contribute to the study of the stability, efficiency, and correctness of decentralized manipulations of complex discrete structures. As a specific example, we propose the *decentralized tree editing* problem for study. The well-known conventional tree (or string) editing problem is to take two trees (strings) and a collection of editing primitives, and determine an optimal sequence of primitives to transform one tree into the second. This abstract problem has practical applications in network management, failure recovery, and software development.

The *decentralized* tree editing problem must be solved by multiple, communicating agents. Changes to both trees may occur asynchronously, and different agents learn of changes at different times, perhaps in different relative orders. They may not share a centralized database, and a locking facility on the trees is strongly discouraged. Agents may not exchange the entire problem, only the minimum needed to coordinate their actions. Ideally, a single agent attempts each necessary operation, and non-conflicting operations are done concurrently by different agents. The solution should permit dynamic addition and removal of agents, as well as tree elements. Further, the solution must be stable, resolving conflicts between agents quickly. These aspects of the decentralized problem must be added to the existing correctness and optimality issues of the conventional problem.

- Explore application-specific impacts on control.

Some applications can do useful work while partitioned or survive the replacement of components without special attention, while others must be explicitly resynchronized when reconfigured, and still others cannot tolerate any visible failures or changes at all.

HPC's mechanisms are sufficient to control the first group. For the latter groups, HPC must be supplemented by mechanisms for manipulating application state, for example, atomic transactions. An agent for an HPC domain must use these other mechanisms correctly. It may be constrained by certain HPC operations, because the application under its control, or the supplementary mechanisms, cannot tolerate the results.

Atomic transactions accommodate the most fragile applications, while real-time applications usually accomodate the harshest environments. However, an application and its environment can make partial allowances for each other, and the resulting systems may prove more efficient than either extreme.

8.3. Reprise

A typical distributed application has a hierarchical structure with well-defined communication patterns between loosely coupled, active computation elements. The distributed environment is an open system composed of autonomous, heterogeneous, asynchronously running sites, subject to independent failure and network partition.

HPC is a study of the use of nested process abstractions and explicit composition to represent such applications. Maintenance, migration, debugging, and adaptation to changing environmental conditions are supported by HPC operations that modify process structure during execution.

The HPC design emphasizes the structural or architectural issues in distributed software, especially interactions involving dynamic reconfiguration, protection, and partition. The contributions of this work come from the detailed consideration of how the seemingly well-known features of abstraction and composition interact with each other and a distributed environment.

This thesis is also a rare case study in consistency control for non-trivial, highly-available services. Operations modifying structure are fully available during network partitions. The inconsistencies that may be encountered during merge have all been identified. Each problem is either avoided, automatically reconciled by the system, or reported to users for application-specific recovery.

Appendix A Formal Description

A. Formal Description

We begin this Appendix with a proof of the characteristic theorem used to express a collection of dynamically changing relations as subsets of a fixed relation. This theorem is critical for our success in treating most dynamic structure as formally immutable and thereby eliminating many sources of inconsistency.

The remaining dynamic effects are modelled by treating each structure as a formal sentence and the operations that transform structures as formal axioms. Section A.2 gives the simplified, abstract form of structure used throughout this Appendix.

The predicates that define legal HPC structures provide this formal system with extensions in a simple model. A legal structure is a true sentence. The constraints that, for example, force strict nesting of objects are all encoded in Section A.3.

Section A.4 defines some core operations (simpler than the primitives provided to clients) and then reduces the client primitives to core operations. Depending upon its arguments, each client primitive may translate into an arbitrary number of core operations (e.g., destruction of a complex subtree).

Every derivation of a *sound* formal system from a true sentence results in a true sentence. A formal system is *complete* in a strong sense if it can derive every true sentence. Space does not permit full proofs of HPC's formal soundness and completeness, but Section A.5 outlines such proofs.

A.1. Characteristic Theorem

HPC increases the number of *formally* immutable properties by distinguishing local knowledge, which may change, from global truths. Many dynamic features of process structure can be limited to creation and destruction of otherwise static elements. We pretend these elements have fixed properties throughout an infinite lifetime, and that we only become aware of them when they are created, and lose awareness of them when they are destroyed. The set of elements known at any time and place is a *characteristic*.

When it is possible to use them, dynamic local characteristics offer a major advantage over dynamic global relations. If the structure visible in a partition is described by a local, completely known relation, and this local relation is formally defined as the intersection of a global (and incompletely known) relation with local, known characteristics, then partitions can be merged simply by taking the set unions of the local relations and the local characteristics.

This property has important consequences. The global sets and relations are *never* needed, only the local ones. This permits complete distributed management of structure. The merge procedure for local relations is extremely trivial, yet guaranteed to preserve the formal definition (and consistency) of the local relations. Finally, there is only a weak constraint governing relations and characteristics. A relation's content is almost irrelevant, so many different structural relationships can be managed this way.

Given

$R \subset X \times Y$

Let R be a binary relation over sets X and Y. These describe the global, immutable (and never completely known) "truth" about structure.

$\text{c-X}_i \subseteq \text{X}\,,\ \text{c-Y}_i \subseteq \text{Y}$

In each partition i, there are local, dynamic characteristic sets, describing the structural elements known at this time and place.

$\text{c-R}_i \doteq \text{R} \cap (\text{c-X}_i \times \text{c-Y}_i)$

A local, dynamic structural relation is formally defined as the intersection of a global relation with the corresponding local characteristics.

$\text{c-X}_i x \wedge \text{R} x\, y \Rightarrow \text{c-Y}_i y$

There is one constraint governing local characteristics and the global relations. The image of a characteristic by the relation must also be characteristic.[12] This is a formal way of saying that we must understand the answers to any questions we can pose.

Theorem

$\text{c-R}_i \cup \text{c-R}_j \equiv \text{R} \cap ((\text{c-X}_i \cup \text{c-X}_j) \times (\text{c-Y}_i \cup \text{c-Y}_j))$

The formal statement of the theorem for binary relations and two partitions. Merging two local relations is identically the fresh intersection of the global relation with the merged local characteristics.

Proof

To simplify the proof we introduce some abbreviations, and freely mix relational, set and predicate notations. No confusion should result. The extensions to n-place relations, and arbitrary numbers of partitions are completely straightforward.

$A \doteq \text{R} x\, y \qquad B \doteq \text{c-X}_i x \qquad C \doteq \text{c-Y}_i y \qquad D \doteq \text{c-X}_j x \qquad E \doteq \text{c-Y}_j y$

First direction: $(\text{c-R}_i \cup \text{c-R}_j) x\, y = (\text{R} \cap ((\text{c-X}_i \cup \text{c-X}_j) \times (\text{c-Y}_i \cup \text{c-Y}_j))) x\, y$

1)	$(\text{c-R}_i \cup \text{c-R}_j) x\, y$	assume
2)	$[A \wedge B \wedge C] \vee [A \wedge D \wedge E]$	1, definition
3)	$A \wedge (B \vee D) \wedge (C \vee E) \wedge [(B \vee E) \wedge (C \vee D)]$	2, deMorgan
4)	$A \wedge (B \vee D) \wedge (C \vee E)$	3, ∧-E
5)	$A \wedge (\text{c-X}_i \cup \text{c-X}_j) x \wedge (\text{c-Y}_i \cup \text{c-Y}_j) y$	4, definition
6)	$(\text{R} \cap ((\text{c-X}_i \cup \text{c-X}_j) \times (\text{c-Y}_i \cup \text{c-Y}_j))) x\, y$	5, definition

Other direction: $(\text{R} \cap ((\text{c-X}_i \cup \text{c-X}_j) \times (\text{c-Y}_i \cup \text{c-Y}_j))) x\, y \Rightarrow (\text{c-R}_i \cup \text{c-R}_j) x\, y$

7)	$A \wedge (B \vee D) \wedge (C \vee E)$	assume, def
8)	$(A \wedge B) \vee (A \wedge D)$	7, ∧-E, deM
9)	$A \dot{\wedge} B \Rightarrow C$	given
10)	$C \vee D$	8, 9, MP, deM, ∧-E
11)	$A \wedge D \Rightarrow E$	given
12)	$B \vee E$	8, 11, MP, deM, ∧-E
13)	$A \wedge (B \vee D) \wedge (C \vee E) \wedge [(B \vee E) \wedge (C \vee D)]$	7, 10, 12, ∧-I
14)	$(\text{c-R}_i \cup \text{c-R}_j) x\, y$	deM, def

QED

[12] This constraint is asymmetric. Only one domain of the relation has to take the role of X. For n-place relations with $n > 2$, this domain can constrain the others either directly, or transitively ($\text{c-X}_i x \wedge \text{R} x\, y\, z \Rightarrow \text{c-Y}_i y$; $\text{c-Y}_i y \wedge \text{R} x\, y\, z \Rightarrow \text{c-Z}_i z$).

A.2. Formal Structure

HPC formal structure can be divided into core and derivative relations, and into immutable and dynamic relations. The core relations are the ones directly manipulated by HPC primitive operations, while the derivative relations describe the more complex consequences of simple operations. In this Section we use PROLOG to define the derivative relations in terms of the core.

The immutable relations describe fixed properties for which inconsistencies can never arise. Normally, only implementation-specific constants would be core and immutable, but this subjects too much structure to merge inconsistencies (Chapter 6). When the characteristic theorem is exploited, only three core dynamic relations are formally manipulated by HPC operations or require non-trival reconciliation after network merges.

A.2.1. Core Immutable Relations

Primitive Structural Elements

```
view(V)
domain(D)
```

Shells, splices, interfaces, connections, controllers, and protection boundaries are all formally reduced to relations on views and domains. These relations are only known partially, through the use of local characteristic sets of views and domains. There is a reserved domain `root`.

```
system(D)
```

Processes, controllers, and the `root` domain form a distinguished subset of all domains.

Primitive IPC Properties

IPC properties are known globally and immutably.

```
medium(M)
  /* M is a supported IPC mechanism */
orientation(O)
  /* M is a supported IPC direction */
```

The supported mechanisms and directions are implementation dependent, but must include the reserved mechanism `control`.

```
reverse(O1, O2)
  /* O1 and O2 are complementary orientations */
```

This relation must be symmetric and pseudo-transitive (odd length sequences must be closed under the relation). This ensures that end-to-end chains that are locally complementary on every link have complementary endpoints.

```
structure([T, P, [M, O]])
structure([T, P, [S, S, ...]])
  /* T is "simple", "bundle", "multiplex", or "multicast"
   * P is "endpoint", "extension", or "masked"
   * medium(M), orientation(O), structure(S)
   */
```

A simple view structure specifies an IPC mechanism and orientation, while a complex structure specifies a sequence of structures. We omit the detailed constraints on the global and immutable `structure` relation (e.g., child structures of a masked complex structure must be also masked) except where the constraints are relevant to later discussion.

Primitive Policy Elements

```
loss(L)
  /* L is "temporary" or "permanent" */
policy(P)
  /* P is "abdicate", "suspend" or "die" */
```

The policy sequences discussed in the text complicate formal proofs without adding much content, so we will limit formal discussion to single basic policies, rather than sequences. Policy elements are known globally and immutably.

View Hierarchies

View hierarchies are only known locally, based on the characteristic view.

```
component(C, P)
  /* C is a component view of P */
```

Every view has exactly one parent throughout its lifetime, except the roots of view hierarchies that never have a parent. Roots of view hierarchies are the bundle endpoints comprising shells, and their immediate children are the interfaces presented to clients.

Protection and Privilege

```
member(V, D)
  /* view V is a member of domain D */
```

A view belongs to exactly one domain throughout its lifetime. It is replaced with a different view (renamed) whenever it would intuitively move between domains. Domain membership is only known locally, based on both characteristic sets.

```
controller(V)
  /* V is top-level view inside controller space */
```

This is the ancestor of the views where agent invocations are received, and HPC system responses are sent.

Primitive Private Peers

```
bound(V1, V2)
  /* V1 and V2 comprise a shell or a splice */
```

The root of a view hierarchy has exactly one private peer throughout its lifetime. These pairs are the formal shells and splices. A root view is replaced with a different view (renamed) whenever its peer would be changed (e.g., by transfer between domains or by endpoint/extension promotion).

```
points_up(V)
  /* V is lower member of a shell */
```

The hierarchy is defined by directed shells. The views that lead toward the root are distinguished.

(This is an immutable property because eversion replaces a directed shell with a splice. Eversion could safely preserve view identifiers by making points_up a dynamic property *provided* changes to the relation are monotonic. Eversion satisfies this condition: false is never changed to true.)

Additional IPC properties

Some IPC properties are known only locally, through the view characteristic.

```
vstruct(V, S)
  /* view V has structure S */


viable(V)
  /* view V is a complex endpoint not in a process space,
   *        or a simple endpoint in a process space
   */
```

Viable views are those that can be used for communication. This is the only (indirect) reflection of real processes in this formal model.

```
index(V, I)
  /* fixed number to determine corresponding components */
```

A small integer based on view structure, or a unique number based on invocations of new.

A.2.2. Derivative Immutable Relations

```
simple(V)     :- vstruct(V, [simple,    _, _]).
bundle(V)     :- vstruct(V, [bundle,    _, _]).
multicast(V)  :- vstruct(V, [multicast, _, _]).
multiplex(V)  :- vstruct(V, [multiplex, _, _]).

extension(V)  :- vstruct(V, [_, extension, _].
endpoint(V)   :- vstruct(V, [_, endpoint,  _].
masked(V)     :- vstruct(V, [_, masked,    _].
```

These predicates simply provide easier access to the vstruct relation.

```
corresponding(V1, V2) :- index(V1, I), index(V2, I).
```

Correspondence follows directly from the fixed index.

```
static_component(C, P) :- component(C, P), bundle(P), endpoint(P).
static_component(C, P) :- component(C, X), static_component(X, P).
```

Creating a bundle endpoint requires automatic creation of its immediate children. These are static, as opposed to dynamic, component views.

```
complementary([simple, _, [M, O1]], [simple, _, [M, O2]]) :-
  reverse(O1, O2).
complementary([C, _, S1], [C, _, S2]) :-
  complementary(S1, S2).
complementary([S1h | S1t], [S2h | S2t]) :-
  complementary(S1h, S2h), complementary(S1t, S2t).
```

Peer views must have complementary structure. For simple views the medium must be the same, and the orientations must be complementary. For complex views, the component structure(s) must be complementary. Multicast and multiplex views have a single structure, while bundle views have a list of structures each of which must be complementary.

```
splice(V1, V2) :- bound(V1, V2), not(points_up(V1) ; points_up(V2)).
shell(V1, V2)  :- bound(V1, V2),    (points_up(V1) ; points_up(V2)).
```

One side of a shell points leads toward the root. This is the only formal distinction between shells and splices.

```
descendant(D, A) :- component(D, A).
descendant(D, A) :- component(D, B), descendant(B, A).
```

```
toplevel(V) :- not(component(V, _)).
```

These predicates just provide easier access to the component relation.

A.2.3. Core Dynamic Relations

Spaces

```
adjacent(V1, V2)
  /* V1 and V2 are currently adjacent view hierarchy roots */
```

It is technically more convenient to maintain the adjacency of view hierarchy roots, rather than all views. Spaces are defined as the equivalence classes of a relation over all views, samespace, derived from this core relation. All the views in a class are incident on the same space. Two spaces are merged by making their view hierarchies all adjacent to one another.

Merge inconsistencies can violate the equivalence constraint. These violations are reconciled automatically.

Public Peers

```
connected(V1, V2)
  /* V1 and V2 are currently private peers */
```

This is always a symmetric and anti-reflexive relation. Ideally, it relates distinct pairs of views, but merge inconsistencies can violate that constraint. Clients must resolve these violations.

Terminal Policies

```
loss_policy(D, L, P)
  /* P is the policy for loss of control L over domain D */
```

In the complete HPC system, P is a non-empty list of basic policies or animations with parameters. The temporary list is terminated with **suspend**. The permanent list may not contain **suspend** and is terminated with **abdicate**. In this formal appendix, P must be a single basic policy.

Merge inconsistencies are resolved automatically. This formal model does not capture the sequential application of terminal policies.

A.2.4. Derivative Dynamic Relations

Spaces

```
samespace(V1, V2) :- adjacent(V1, V2).
samespace(V1, V2) :- component(V1, P), samespace(P, V2).
samespace(V1, V2) :- component(V2, P), samespace(V1, P).

clear(V1, V2)  :- shell(V1, V2), member(V1, D), member(V2, D).

opaque(V1, V2) :- bound(V1, V2), member(V1, D1), member(V2, D2), D1 != D2.
opaque(V1, V2) :- splice(V1, V2).

real_boundary(V, D) :- member(V, D), bound(V, V1), not(member(V1, D)).

boundary(V, D)        :- member(V, D), opaque(V, _).
```

```
below(V1, V2) :- adjacent(V1, V3), points_up(V3),
                 clear(V3, V4), adjacent(V2, V4).
below(V1, V2) :- below(V1, V), below(V, V2).

superior(V, D) :- not(below(V, _)), boundary(V, D).
inferior(V, D) :- not(below(_, V)), boundary(V, D).
```

View Suspension

```
suspended(V) :- connected(V, V1), connected(V, V2), V1 != V2.
suspended(V) :- superior(V, D), superior(V1, D), V1 != V.
suspended(V) :- boundary(V, D), policy(D, temporary, suspend),
                temp_control_loss(D).
suspended(V) :- component(V, P), suspended(P).
```

suspended describes forced suspensions due to violations of constraints discussed in the next Section, or to temporary loss of control. Four conditions force the suspension of a view. It may have multiple connections, be the root view of a surplus superior domain boundary, be a domain boundary for a suspended domain, or be a child of a suspended view. Forcibly suspending one view may indirectly affect the liveness of other views through other relations.

```
temp_control_loss(D) :- control_view(C, D), liveness(C, suspended).
temp_control_loss(D) :- control_view(C, D), liveness(C, dead),
  real_boundary(B, D), chain([private, C], [public, B], _).

control_view(V, D) :-
  member(C, D), controller(C), component(P, C), component(V, P).
```

A domain is suspended when a temporary loss of control is detected and the current temporary terminal policy is suspended. A controller shell is a pair of bundles with one multicast endpoint component. Inside the shell, the HPC system creates one control endpoint component of the multicast view (two levels down from the shell). If this internal control view is suspended, or dead with at least one chain reaching the domain boundary, control has been lost at least temporarily.

Private Peers and Chains

```
private(V1, V2) :- bound(V1, V2).
  /* V1 and V2 are directly bound private peers */ private(V1, V2) :-
  component(V1, P1 , component(V2 [illegible]), corresponding(V1, V2),
  chain([private, P1], [private [illegible]]). /* V1 and V2 are
  corresponding components of pr[illegible] peers */

chain([private, V1], [private, V2], [[]]) :- private(V1, V2).
chain([public,  V1], [public,  V2], [[]]) :- connected(V1, V2).
  /* V1 and V2 are bound in one step, privately or publically */

chain([private,  V1], T, [[public,  V2] | R]]) :-
  private(V1, V2),    chain([public,  V2], T, R).
  /* A private one step binding from V1 to V2
   * extends chains starting with a public binding of V2
   *
chain([public,   V1], T, [[private, V2] | R]]) :-
  connected(V1, V2), chain([private, V2], T, R).
  /* A public one step binding from V1 to V2
   * extends chains starting with a private binding of V2
   *
```

Private peers and chains are defined recursively in terms of corresponding components and direct private and public bindings. The definition of private peers given here is the looser one used in the path maintenance algorithm.

(A serious PROLOG implementation would require a check for cycles in the last two clauses of chain/3.)

Liveness

```
liveness(V, suspended) :- suspended(V), !.

liveness(V, alive)      :-
  private(V, V2), viable(V2), not(suspended(V2)), !.
  /*  [|]
   * V  V2
   */

liveness(V, alive)      :-
  chain([private, V], [public,  V3], [[public, V2]]),
  liveness(V3, alive), !.
  /*  [|]==[|]
   * V  V2 V3
   */

liveness(V, suspended) :-
  private(V, V2), viable(V2), suspended(V2), !.

liveness(V, suspended) :-
  chain([private, V], [public,  V3], [[public, V2]]),
  liveness(V3, suspended), !.

liveness(V, dead).
```

A view can be forced to suspended liveness. Otherwise, we look down chains starting with a private binding, first for alive views, then for suspended ones. If no one step private bindings lead to viable endpoints, then liveness is inherited from the next step down the chain. if no viable peers are found, either alive or suspended, the view is dead. (Cycles must be avoided in the third and fifth clauses given here.)

A.3. Legal Structures

The core relations provide a structural framework without any definition of legal structure. The permissible HPC structures form a very small part of the possible ones. Treated as a formal model, the core relations give an alphabet of symbols, with no axioms constraining their interpretations. We formally define the legal HPC structures as those interpretations satisfying the axioms given in this section. These axioms are applied to the structure known in the current partition, that is, the local characteristic relations. The renaming technique violates many axioms when the formally complete (over all time and partitions) global structure is considered.

It will be our goal in the next few sections to show that legal structures are closed under the HPC primitive operations (soundness), and that any legal structure (up to isomorphism) can be created by the primitives (completeness). However, legal structures are *not* closed under merge inconsistencies. After the HPC system applies its automatic reconcilation of merge inconsistencies, two axioms concerning mutable relations can remain violated. In these cases, legal structure must be restored by application managers. We will make no attempt to capture formally the broader sense of consistent behavior enforced over a merge inconsistency by suspended liveness.

A.3.1. Notational Conventions

As in the proof of the characteristic theorem, it is convenient to treat relations as predicates, as sets, and as functions. To use a relation as a predicate, we supply a single value for each domain of the relation in parenthesis.

```
component(v, p)          // true iff v is a component of p
```

When treating relations as sets of tuples, we use conventional brace ({s1, s2}) and angle bracket (<t1, t2>) notation. Modification of structure is expressed using C-like notation for set addition and set subtraction. E.g.,

```
connected += { <v1, v2>, <v2, v1> }

viable    -= { v1 }
```

To treat relations as (partial) functions, we want to provide a value for some domain(s) of the relation, and get back the values of the other domains. For example, we would like to know what views are children of the given view v. By providing a set of input values, we obtain the image of the set under the relation. We introduce a quoted number convention to indicate which domain(s) of the relation is (are) to cast the image.

```
component'2'{v}         // set of children of v
component'1'{v}         // set of parents of v

loss_policy'1,2'{<d,1>} // policies that apply to domain d under loss 1
```

We systematically confuse a singleton set with its member.

Vertical bars denote the cardinality of a set.

```
| { } |  == 0

| { <v1, v2> } | == 1
```

When used as predicates, an empty set denotes falsehood.

Iteration over the members of a set uses this notation.

```
for s in {s1, s2, s3}
  // body of iteration
```

Bundle structures specify a sequence of child structures. We will need to extract this sequence from the bundle and refer to its individual elements. Given a structure s, we use this notation:

```
s.components            // sequence part of structure
|s.components|          // number of elements
s.components[i]         // i-th element of sequence
```

A.3.2. New Element

Structure modification formally involves structure that always existed and simply wasn't known in the current partition. By changing characteristic sets we change the known structure. Actually, we rename and generate new structure on the fly. We use the assertion new(e) to indicate a view or domain that has never been known, and avoid axioms that describe structure before its generation. The rigor required to formalize this meta-axiom is unrewarding.

A.3.3. Immutable Relations

```
domain(d)                                    // characteristic
view(v)                                      // characteristic
```

The local characteristic relations all rely on these characteristic sets, which have no internal constraints.

```
controller(v) => view(v)                    // characteristic
controller(v) => viable(v)                  // liveness
controller(v) => points_up(v)               // root of space
controller(v) => adjacent'1'v == {v}        // leaf space
controller(v) => connected'1'(descendant'2'v) == { }    // empty space
controller(v) => system(member'1'v)         // protected
controller(v) => vstruct(v, {bundle, endpoint,  // structure
                          {multicast, endpoint,
                            {simple, endpoint, {control, in}}}})
```

A controller view must be the inside boundary of an empty shell with a specific structure. They are protected from application domains.

```
component(v, p) => view(v)                  // characteristic
component(v, p) => view(p)                  // characteristic
component(v, p) => complex(p)               // parent structure
component(v, p) => endpoint(p)              // parent structure
component(v, p) => viable(p)                // parent liveness
component(v, p) =>                          // common structure
  bundle(p)     => vstruct'1'v == vstruct'1'p.components[index'2'v]
  multicast(p)  => vstruct'1'v == vstruct'1'p.components[1]
  multiplex(p)  => vstruct'1'v == vstruct'1'p.components[1]
component(v, p) => domain'1'v == domain'1'p     // common domain
component(v, p) => component'1'v == {p}     // unique map
```

The parent of a component view must be a viable complex endpoint. The structure of the component must be one of the structures specified by the parent.

```
member(v, d)     => view(v)                 // characteristic
member(v, d)     => domain(v)               // characteristic
| member'1'v | == 1                         // unique, complete map
```

Every view belongs to exactly one domain.

```
bound(v1, v2)    => view(v1)                // characteristic
bound(v1, v2)    => view(v2)                // characteristic
bound(v1, v2)    => complementary(vstruct'1'v1, vstruct'1'v2)
bound(v1, v2)    => bundle(v1)              // view structure
bound(v1, v2)    => endpoint(v1)            // view structure
bound(v1, v2)    => bound(v2, v1)           // symmetric
bound(v1, v2)    => v1 != v2                // anti-reflexive
bound(v1, v2)    => bound'1'v1 == {v2}      // unique map
bound(v1, v2)    => toplevel(v1)            // only vh roots
```

Shells and splices must be distinct toplevel bundles with complementary structures. HPC also requires a unique private binding, although this constraint could be relaxed.

```
points_up(v)     => view(v)                 // characteristic
points_up(v)     => toplevel(v)             // only vh roots
```

Only toplevel views are part of the hierarchical relation.

```
vstruct(v, s)    => view(v)                 // characteristic
vstruct(v, s)    => structure(s)            // characteristic
vstruct(v, s)    => !masked(v)              // no masked views
| vstruct'1'v | == 1                        // unique, complete map
```

Every view has exactly one structure.

```
viable(v)        => view(v)                 // characteristic
viable(v)        => toplevel(v)             // only vh roots
```

Only toplevel views are part of the viability relation.

```
index(i, v)       => INTEGER(i)                 // characteristic
index(i, v)       => view(v)                    // characteristic
| index'2'v | == 1                              // unique, complete map
```

Every view has exactly one index.

A.3.4. Mutable Relations

```
adjacent(v1, v2)  => view(v1)                   // characteristic
adjacent(v1, v2)  => view(v2)                   // characteristic
adjacent(v1, v2)  => adjacent(v2, v1)           // symmetric
adjacent(v1, v2)  <= adjacent(v1, v3) & adjacent(v3, v2)// transitive
adjacent(v1, v2)  => domain'1'v1 == domain'1'v2 // common domain
adjacent(v, v)    <= toplevel(v)                // all vh roots
adjacent(v1, v2)  => toplevel(v1)               // only vh roots
```

Only toplevel views are part of the adjacency relation, an equivalence relation. Each equivalence class defines a space and must belong to a common domain.

```
connected(v1, v2) => view(v1)                   // characteristic
connected(v1, v2) => view(v2)                   // characteristic
connected(v1, v2) => samespace(v1, v2)          // same space
connected(v1, v2) => extension(v1)              // structure
connected(v1, v2) => extension(v2)              // structure
connected(v1, v2) => complementary(vstruct'1'v1, vstruct'1'v2)
connected(v1, v2) => connected(v2, v1)          // reflexive
connected(v1, v2) => v1 != v2                   // anti-reflexive
connected(v1, v2) => connected'1'v1 == {v2}     // unique map
```

Connected views must be distinct complementary extensions in the same space. HPC allows at most one connection per view, but merge inconsistencies can lead to multiple connections.

```
loss_policy(d, l, p)   => domain(d)             // characteristic
loss_policy(d, l, p)   => loss(l)               // characteristic
loss_policy(d, l, p)   => policy(p)             // characteristic
loss_policy(d, permanent, p) => p != suspend    // must recover control
domain(d) && !system(d) =>                      // unique, complete map
  | loss_policy'1,2'<d,l> | == 1
```

Each non-system domain must have exactly one policy for each type of loss of control. HPC refuses permanent responsibility for a domain. In this presentation, only basic policies are allowed.

A.3.5. Hierarchical Constraints

The axioms which enforce the appearance of a hierarchy are definitely the most complex as well as the hardest to handle when showing soundness.

```
d == root => | superior'2'd | == 0              // root
d != root => | superior'2'd | == 1              // unique superior
```

The overall root space has no upper pointing views, while all other legal spaces have exactly one view hierarchy that points toward the root. However, merge inconsistencies can produce multiple superior interfaces at the overall root at the root of a non-system domain.

```
below(v1, v2) => !below(v2, v1)                 // anti-symmetric
below(v1, v2) => !samespace(v1, v2)             // anti-reflexive
below(v1, v2) && below(v1, v3)                  // hierarchy
  => samespace(v2, v3) || below(v3, v2) || below(v2, v3)

member'1'v1 == member'1'v2                      // contiguous
  => samespace(v1, v2) || below(v1, v2) || below(v2, v1)
```

All spaces of a domain must be contiguous, and they must be organized into a tree.

```
inferior(v, root)
  => vstruct'1'(adjacent'1'v) == {bundle, endpoint, []}

 system(d) && d != root => | inferior'2'd | == 0
!system(d) =>
   | { c : c in opaque'1'(inferior'2'd) && controller(c) } | == 1
```

Inferior domains of the root domain are opaque top level applications with no interfaces. Process and controller domains have no inferior domains. Non-system domains must have a unique, immediately adjacent, controller domain.

A.3.6. Constraints on View Structure

```
structure(s) && s == [multicast, .., seq] => |seq| == 1
structure(s) && s == [multiplex, .., seq] => |seq| == 1
```

Exactly one structure must be specified for dynamically created views.

A.4. Operations on Structure

If the core relations define a domain of structural models, the constraints on legal structures are axioms, and a specific structure is a formal sentence, then operations on structure are formal rules of inference on sentences.

We present HPC operations in three stages. First, we define some auxiliary operations that are not sound when used by themselves. Using these definitions, we present the core structural operations, which are sound when invoked with the appropriate preconditions. Finally, we reduce the operations available to HPC clients into core operations.

A.4.1. Auxiliaries

View Hierarchies

```
vh_create(n, i, d, p, s, v)

let:

preconditions:

effects:
  domain        // no change
  system        // no change
  controller    // no change
  view          += { v }
  component     += if (!empty(p)) then { <v, p> }
  member        += { <v, d> }
  bound         // no change
  points_up     // no change
  vstruct       += { <v, s> }
  viable        += if complex(v) && endpoint(v)  then { v }
  index         += if multiplex(p) then { <n, v> } else { <i, v> }

  adjacent      // no change
  connected     // no change
  loss_policy   // no change

  if bundle(v) && endpoint(v)
    for ci in { 1, ..., |s.components| }
      new(c)
      vh_create(n, ci, d, v, s.components[ci], c)


vh_destroy(i, d, p, s, v)

let:
  pu = points_up(v)
  vi = viable(v)

preconditions:

effects:
  for cv in component'2'v
    let ci = index'1'cv
    let si = vstruct'1'cv
    vh_destroy(ci, d, v, si, cv)

  domain        // no change
  system        // no change
  controller    // no change
  view          -= { v }
  component     -= if (!empty(p)) then { <v, p> }
  member        -= { <v, d> }
  bound         // no change
  points_up     // no change
  vstruct       -= { <v, s> }
  viable        -= if vi then { v }
  index         -= { <v, i> }

  adjacent      // no change
  connected     // no change
  loss_policy   // no change
```

Renaming

```
vn_rename(do, dn, vo, vn)

let:
  po == component'1'vo
  pn == component'1'vn

  s  == structure'1'vo
  i  == index'1'vo

  cv == connected'1'vo

preconditions:

effects:
  domain        // no change
  system        // no change
  controller    // no change
  bound         // no change
  points_up     // no change
  viable        // no change

  adjacent      // no change
  loss_policy   // no change

  view          += { vn }
  component     += if (!empty(pn)) then { <vn, pn> }
  member        += { <vn, dn> }
  vstruct       += { <vn, s> }
  index         += < vn, i >

  connected     += (cv X {vn})

  for co in c
    new(cn)
    vn-rename(do, dn, co, cn)

  view          -= { vo }
  component     -= if (!empty(po)) then { <vo, po> }
  member        -= { <vo, do> }
  vstruct       -= { <vo, s> }
  index         -= < vo, i >

  connected     -= (cv X {vo})
```

```
sh_rename(dot, dob, dnt, dnb, to, bo, tn, bn)

let:
  vi1 == viable(to)
  vi2 == viable(bo)
  pu1 == points_up(to)
  pu2 == points_up(bo)

  a1  == adjacent'1'to
  a2  == adjacent'1'bo

preconditions:
  bound(to, bo)

effects:

  domain        // no direct change
  system        // no direct change
  controller    // no direct change
  view          // no direct change
  component     // no direct change
  vstruct       // no direct change
  index         // no direct change

  loss_policy   // no direct change

  bound         -= { <to, bo>, <bo, to> }
  points_up     -= if pu1 then to
  points_up     -= if pu2 then bo
  viable        -= if vi1 then {to}
  viable        -= if vi2 then {bo}

  adjacent      -= if !empty(a1) (a1 X {to}) + ({to} X a1) + {<to, to>}
  adjacent      -= if !empty(a2) (a2 X {bo}) + ({bo} X a2) + {<bo, bo>}

  vh_rename(do1, dn1, to, tn)
  vh_rename(do2, dn2, bo, bn)

  bound         += { <tn, bn>, <bn, tn> }
  points_up     += if pu1 then tn
  points_up     += if pu2 then bn
  viable        += if vi1 then {tn}
  viable        += if vi2 then {bn}

  adjacent      += if !empty(a1) (a1 X {tn}) + ({tn} X a1) + {<tn, tn>}
  adjacent      += if !empty(a2) (a2 X {bn}) + ({bn} X a2) + {<bn, bn>}

do_rename(do, dn, to, bo, tn, bn)

let:

preconditions:

effects:
  let db == member'1'bo
  sh_rename(dc, db, to, bo, tn, bn)

  if clear(to, bo)
    for cot in adjacent'1'bo - {bo}

      new(cnt)
      new(cnb)
      let cob == bound'1'cot
      do_rename(do, dn, cot, cob, cnt, cnb)
```

A.4.2. Core Operations

Connections

```
c_create(v1, v2)

let:

preconditions:

effects:
  domain        // no change
  system        // no change
  controller    // no change
  view          // no change
  component     // no change
  member        // no change
  bound         // no change
  points_up     // no change
  vstruct       // no change
  viable        // no change
  index         // no change

  adjacent      // no change
  connected     += { <v1, v2>, <v2, v1> }
  loss_policy   // no change


c_destroy(v1, v2)

let:

preconditions:

effects:
  domain        // no change
  system        // no change
  controller    // no change
  view          // no change
  component     // no change
  member        // no change
  bound         // no change
  points_up     // no change
  vstruct       // no change
  viable        // no change
  index         // no change

  adjacent      // no change
  connected     -= { <v1, v2>, <v2, v1> }
  loss_policy   // no change
```

```
c_clear(v

let:
  vs == descendants'2'(adjacent'1'v

preconditions:

effects:
  domain        // no change
  system        // no change
  controller    // no change
  view          // no change
  component     // no change
  member        // no change
  bound         // no change
  points_up     // no change
  vstruct       // no change
  viable        // no change
  index         // no change

  adjacent      // no change
  connected     -= (vs X vs)
  loss_policy   // no change
```

Shells

```
s_split(d, vu, su, vsu, vl, sl, vsl)

let:

preconditions:
  adjacent'l'vsu == vsu - vsl
  adjacent'l'vsl == vsu - vsl

  new(n)

effects:
  vh_create(n, 0, d, { }, sl, vl)
  vh_create(n, 0, d, { }, su, vu)

  domain        // no change
  system        // no change
  controller    // no change
  view          // no direct change
  component     // no direct change
  member        // no direct change
  bound         += { <vu, vl>, <vl, vu> }
  points_up     += { vu }
  vstruct       // no direct change
  viable        += { vu, vl }
  index         // no direct change

  adjacent      += (vsu X {vu}) + ({vu} X vsu) + { <vu, vu> }
  adjacent      += (vsl X {vl}) + ({vl} X vsl) + { <vl, vl> }
  adjacent      -= (vsu X vsl)  + ( vsl X vsu)
  connected     // no change
  loss_policy   // no change


s_merge(d, vu, su, vsu, vl, sl, vsl)

let:

preconditions:

effects:
  domain        // no change
  system        // no change
  controller    // no change
  view          // no direct change
  component     // no direct change
  member        // no direct change
  bound         -= { <vu, vl>, <vl, vu> }
  points_up     -= { vu }
  vstruct       // no direct change
  viable        -= { vu, vl }
  index         // no direct change

  adjacent      -= (vsu X {vu}) + ({vu} X vsu) + { <vu, vu> }
  adjacent      -= (vsl X {vl}) + ({vl} X vsl) + { <vl, vl> }
  adjacent      += (vsu X vsl)  + ( vsl X vsu)
  connected     // no change
  loss_policy   // no change

  vh_destroy(0, d, { }, su, vu)
  vh_destroy(0, d, { }, sl, vl)
```

Views

```
v_create(d1, n, s1, p1, c1)

let:

preconditions:

effects:
  vh_create(n, l, d1, p1, s1, c1);

  for pr in private'l'p1
    let dr == member'l'pr
    let sr == vstruct'l'pr
    if viable(pr)
      new(cr)
      vh_create(n, l, dr, pr, sr.components[1], cr)


v_destroy(d, n, s, p, c)

let:

preconditions:

effects:
  vh_destroy(l, d, p, s, c
```

Processes

```
p_create(d0, d1, ro0, ro1, m0, m1)

let:

preconditions:

effects:
  oc_rename(d0, d1, ro0, ro1, m0, m1)

  domain        += { d1 }
  system        += { d1 }
  controller    // no change
  view          // no direct change
  component     // no direct change
  member        // no direct change
  bound         // no direct change
  points_up     // no direct change
  vstruct       // no direct change
  viable        += { m1 }
  index         // no direct change

  adjacent      // no direct change
  connected     // no direct change
  loss_policy   // no change
```

```
p_destroy(du, dl, rou, rol, mu, ml)

let:

preconditions:

effects:
  do_rename(dl, du, rou, rol, mu, ml)

  domain       -= { dl }
  system       -= { dl }
  controller   // no change
  view         // no direct change
  component    // no direct change
  member       // no direct change
  bound        // no direct change
  points_up    // no direct change
  vstruct      // no direct change
  viable       -= { ml }
  index        // no direct change

  adjacent     // no direct change
  connected    // no direct change
  loss_policy  // no change
```

Domains

```
d_split(d, rot, rob, rmt, rmb, cot, cob)

let:

preconditions:
  new(dc)
  new(dn)
  new(cmt)
  new(cmb)

effects:
  domain         += { dn, dc }
  system         += { dc }
  controller     += { cob }
  view           // no direct change
  component      // no direct change
  member         // no direct change
  bound          // no direct change
  points_up      // no direct change
  vstruct        // no direct change
  viable         += { cob }
  index          // no direct change

  adjacent       // no direct change
  connected      // no direct change
  loss_policy    += {<dn, temporary, suspend>, <dn, permanent, abdicate>}

  do_rename(d, dc, cot, cob, cmt, cmb)
  do_rename(d, dn, rot, rob, rmt, rmb)


d_merge(d, rot, rob, rmt, rmb, cot, cob)

let:
  p  == policy'1'd
  dn == member'1'rot
  dc == member'1'cot

preconditions:
  new(cmt)
  new(cmb)

effects:
  do_rename(dc, d, cot, cob, cmt, cmb)
  do_rename(dn, d, rot, rob, rmt, rmb)

  domain         -= { dn, dc }
  system         -= { dc }
  controller     -= { cob }
  view           // no direct change
  component      // no direct change
  member         // no direct change
  bound          // no direct change
  points_up      // no direct change
  vstruct        // no direct change
  viable         -= { cob }
  index          // no direct change

  adjacent       // no direct change
  connected      // no direct change
  loss_policy    -= ({d} X p)
```

Structure Hierarchies

```
t_kill(du, vu, vl, nu, nl)

let:
  su = vstruct'l'vu
  vl = bound'l'vu
  dl = member'l'vl
  sl = vstruct'l'vl

preconditions:
  !controller(vl)

effects:
  if splice(vu, vl)
    new(xl)
    new(xu)
    S_destroy(vu, vl, nu, nl, xl, xu)

  if inferior(vu, du) && system(dl)
    p_destroy(du, dl, vu, vl, nu, nl)

  if inferior(vu, du) && !system(dl)
    new(xu)
    new(xl)
    d_merge(du, vu, vl, xu, xl, cu, cl)
    t_kill(du, xu, xl, nu, nl)

  if clear(vu, vl)
    c_clear(vl)
    for cu in adjacent'l'vl - {vl}
      let cl  = bound'l'cu
      let scu = vstruct'l'cu
      let scl = vstruct'l'cl
      c_clear(cu)
      t_kill(dl, cu, cl, nu, nl)
      s_merge(du, nu, scu, adjacent'l'nu - {nu},
                  nl, scl, adjacent'l'nl - {nl})
```

A.4.3. HPC Primitives

Connections

```
connect(d, v1, v2)

let:
  s1 == vstruct'1'v1
  s2 == vstruct'1'v2

preconditions:
  view(v1)                                        type
  view(v2)

  member(v1, d)                                   privilege
  member(v2, d)

  samespace(v1, v2)                               local composition

  extension(s1)                                   view structure
  extension(s2)

  complementary(s1, s2)

  connected'1'v1 = { }                            not connected
  connected'1'v2 = { }

effects:
  c_create(v1, v2)


disconnect(d, v1, v2)

let:
  s1 == vstruct'1'v1
  s2 == vstruct'1'v2

preconditions:
  view(v1)                                        type
  view(v2)

  member(v1, d)                                   privilege
  member(v2, d)

  connected(v1, v2)                               mutually connected

effects:
  c_destroy(v1, v2)
```

Shells

```
enclose(d, vs1, su, sl)

let:
  views = adjacent'1'vs1
  vs2   = views - vs1
  vsu   = if (points_up'1'vs1) then vs1 else vs2
  vsl   = if (points_up'1'vs1) then vs2 else vs1

preconditions:
  for v member vs1                            argument type
    view(v)

  structure(su)
  structure(sl)

  for v in vs1                                privilege
    member(v, d)

  complementary(su, sl)                       view structure

  !empty(vsu)                                 partition of space
  for v1 in vs1
    for v2 in vs1
      adjacent(v1, v2)

  connected'1'(descendant'2'vsu) subset vsu   composition unaffected
  connected'1'(descendant'2'vsl) subset vsl

  new(vu)
  new(vl)

effects:
  s_split(d, vu, su, vsu, vl, sl, vsl)


disclose(d, vu, vl)

let:
  vsu = adjacent'1'(vu) - (vu)
  vsl = adjacent'1'(vl) - (vl)
  su  = vstruct'1'vu
  sl  = vstruct'1'vl

preconditions:
  view(vu)                                    argument type
  view(vl)

  member(vu, d)                               privilege
  member(vl, d)

  shell(vu, vl)                               merger of spaces

  connected'1'(descendant'2'(vu)) = { }       composition unaffected
  connected'1'(descendant'2'(vl)) = { }

effects:
  s_merge(d, vu, su, su, vl, sl, vsl)
```

Views

```
new(c, p)

let:
  s == vstruct'1'p

preconditions:
  view(p)                                        argument type

  endpoint(p)                                    view structure
  multiplex(p) || multicast(p)

  member(p, c)                                   privilege

  new(n)
  new(c)

effects:
  v_create(d, n, p, s.components[1], c)


delete(d, c)

let:
  p == component'1'c
  s == vstruct'1'c
  i == index'2'c

preconditions:
  view(c)                                        argument type

  multiplex(p) || multicast(p)                   view structure

  member(c, d)                                   privilege

  connected'1'(descendant'2'c) = { }             unconnected

effects:
  v_destroy(d, i, s, p, c
```

Process Manipulation

```
animate(du, vu, vl, su, sl, i)

let:
  mco == [simple, endpoint, [control, out]]
  mci == [simple, endpoint, [control, in]]

  scu == [bundle, endpoint, [multicast, endpoint, mco]]
  scl == [bundle, endpoint, [multicast, endpoint, mci]]

preconditions:
  view(vu)                                        type
  view(vl)

  member(vu, d)                                   privilege
  member(vl, d)

  adjacent'1'vl == { vl }                         empty leaf
  connected'1'(descendant'2'vl) == { }

  if su != { }                                    view structure
    complementary(su, sl)
    i <= |su.components|

  new(dl)
  new(nu)
  new(nl)

effects:
  if su == { }
    p_create(du, dl, vu, vl, nu, nl)    // create simple domain

  if su != { }
    new(cu)                             // create leaf for controller
    new(cl)
    s_split(du, cu, scu, {vl}, cl, scl, { })
    let ce == component'2'cu            // create multicast view
    new(cm)
    v_create(du, 1, mco, ce, cm)
    new(xu)                             // create leaf for manager
    new(xl)
    s_split(du, xu, su, {vl, cu}, xl, sl, { })
    let me == { v : component(v, xu) && index(i, v) } // connect manager
    let ms == vstruct'1'me
    c_create(me, cm)
    new(cp)                             // create process in leaf
    new(yu)
    new(yl)
    p_create(du, cp, xu, xl, yu, yl)
    d_split(du, dl, yu, yl, nu, nl)     // create a complex domain
```

```
kill(dv, vu)

let:
  vl == bound'1'vu
  cu == { v : boundary(v, dv) && bound(v, cu) && controller(c) }
  cl == bound'1'cu
  rl == superior'2'cu
  ru == bound'1'rl
  dr == member'1'ru

preconditions:
  view(vu)                                        type

  member(vu, dv)                                  privilege

  new(nu)
  new(nl)

effects:
  if below(cu, vl) || samespace(cu, vl)
    if policy'1,2'<dv, permanent> == abdicate
      d_merge(dr, ru, rl, nu, nl, cu, cl)

    if policy'1,2'<dv, permanent> == die
      t_kill(dr, ru, rl, nu, nl)

  if !(below(cu, vl) || samespace(cu, vl))
    t_kill(dv, vu, vl, nu, nl)


die(dl, vl)

let:
  vu == bound'1'vl
  sl == vstruct'1'vl
  du == member'1'vu

preconditions:
  view(vl)                                        type

  superior(vl, dl)                                privilege

  new(nu)
  new(nl)

effects:
  t_kill(du, vu, vl, nu, nl)
```

Domain Manipulation

```
invest(d, rot, cnot)

let:
  rob  = bound'1'rot
  cnob = bound'1'cnot
  cot  = { v : boundary(v, d) && bound(v, c) && controller(c) }
  cob  = bound'1'cot

preconditions:
  view(rot)                                 argument type
  view(cnob)

  member(rot, d)                            privilege
  member(cnot, d)

  clear(rot, rob)                           root white box
  clear(cnot, cnob)                         controller white box

  points_up(rob)
  points_up(cnob)

  !(below(cot, rob) || samespace(cot, rob))     d keeps old controller
   below(cnob, rob)                          dn gets new controller

  adjacent'1'cnob = { cnot }                empty leaf
  connected'1'(descendant'2'cnob) = { }

  vstruct(cnob, [bundle, endpoint,          view structure
                [multicast, endpoint,
                  [simple, endpoint, [control, in]]]])
  new(rnt)
  new(rnb)

effects:
  d_split(d, rot, rob, rnt, rnb, cnot, cnob)


depose(d, rot)

let:
  rob  = bound'1'rot
  do   = member'1'rob
  cot  = { v : boundary(v, do) && bound(v, c) && controller(c) }
  cob  = bound'1'cot

preconditions:
  view(rot)                                 argument type

  inferior(rot, d)                          privilege, black box

  new(rnt)
  new(rnb)

effects:
  if !splice(rot, rob) && policy'1,2'<do, permanent> == abdicate
    d_merge(d, rot, rob, rnt, rnb, cot, cob)

  if  splice(rot, rob) || policy'1,2'<do, permanent> == die
    t_kill(d, rot, rob)
```

```
abdicate(c, rob)

let:
  rot  == bound'1'rob
  cr   == member'1'rob
  cot  == { v : boundary(v, c) && bound(v, c) && controller(c) }
  cob  == bound'1'cot

preconditions:
  view(rob)                                 argument type

  superior(rot, c)                          privilege, black box

  new(rnt)
  new(rnb)
effects:
  if !splice(rot, rob) && policy'1,2'<dc, permanent> == abdicate
    d_merge(cr, rot, rob, rnt, rnb, cot, cob)

  if  splice(rot, rob) || policy'1,2'<dc, permanent> == die
    t_kill(cr, rot, rob)
```

Splices

```
splice(d, vol, vor)

let:
  vll = bound'1'(vol)
  vnl = prespliced(vol, vor)
  vnr = prespliced(vor, vol)
  inl = intermediate(vol, vor)
  inr = intermediate(vor, vol)
  sol = structure'1'(vol)
  sll = structure'1'(vll)
  sor = structure'1'(vor)
  va  = adjacent'1'(vol)

  co  = components'2'(vol)
  cl  = components'2'(vll)

preconditions:
  view(vol)                                 argument type
  view(vll)
  view(vor)

  member(d, vol)                            privilege
  member(d, vll)

  complementary(sol, sor)                   view structure

  adjacent'1'(vll) = { vll }                empty leaf
  connected'1'(descendant'2'(vll)) = { }

  new(ds)
  new(xl)
  new(xr)
effects:
  if inl in view
    S_create(vnl, vnr, vol, vll, inl, inr)

  if !inl in view
    s_split(ds, xl, sol, { }, xr, sll, { })
    S_create(inl, inr, vol, vll, xl, xr)

notes:
  creates miniroots
```

A.5. Soundness and Completeness

A.5.1. Soundness

A full proof of soundness is a simple, but quite laborious task. There are 12 core operations, and about 50 constraints on 14 core relations that must be preserved. Many of the 600-odd individual proofs are trivial, but a large number require non-trivial inference.

The proofs for connect and disconnect are especially simple. They affect only the connected relation, and only by adding or removing a symmetric pair of tuples. Their preconditions immediately establish seven of the nine constraints on connected. Adding a symmetric pair establishes the final two.

Actually, as presented in this Chapter, splice is not sound. The following analysis is typical of the more interesting (dis)proofs. To be unintrusive, splice of a and b does not rename b. After splicing, b's peer is in the domain a was in. A view is a member of one domain during its whole lifetime, and a view has the same private peer during its whole lifetime. Therefore, b's peer must have been in a's domain before splicing. All spaces of a domain are contiguous, but b's peer was not in the domain before splicing: contradiction. One of the intermediate views is also created with no superior views, in a domain other than root. In the HPC implementation these unsoundnesses are avoided by using a more flexible (and complicated) constraint on domain contiguity.

A.5.2. Completeness

Completeness up to isomorphism (trivial relabelling) can be shown in a more elegant fashion. First, observe that every structure can be reduced to the null structure by a sequence of core operations. A further, non-trivial observation is that each such sequence has an inverse (up to isomorphism). Therefore, any true sentence can be derived from any other.

Establishing inverses is non-trivial for two reasons. Most core operations do not have exact inverse operation. For example, **new** may create components of remote views, while **delete** always destroys exactly one view. Fortunately, every core operation has an inverse sequence of operations. A harder problem is that not every sequence of core operations can be produced by a sequence of HPC primitives, and it is necessary to demonstrate that the effects of any primitive can be undone by a sequence of primitives.

Appendix B Bibliography

B. Bibliography

References

[ABB86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, Mach: A New Kernel Foundation for UNIX Development, *Proceedings Summer 1986 USENIX Technical Conference and Exhibition*, Pittsburgh, Pennsylvania, June 1986.

[ABL85] G. T. Almes, A. P. Black, E. D. Lazowska and J. D. Noe, The Eden System: A Technical Review, *IEEE Transactions on Software SE-11*, 1 (January 1985), 43-59.

[Ary81] A. K. Arya, Super: Encapsulated Autonomous Distributed Computations on an Abstract Architecture, Ph.D. Thesis, University of Rochester, July 1981.

[BRT84] F. Baiardi, L. Ricci, A. Tomasi and M. Vanneschi, Structuring Processes for a Cooperative Approach to Fault-Tolerant Distributed Computing, *Proceedings 4th Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, 12-17 October 1984, 218-231.

[BFL76] J. E. Ball, J. A. Feldman, J. R. Low, R. Rashid and P. Rovner, RIG, Rochester's Intelligent Gateway: System Overview, *IEEE Transactions on Software Engineering SE-2*, 4 (1976), 321-328.

[BeG81] P. A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, *Computing Surveys 13*, 2 (June 1981), 185-222.

[BeG84] P. A. Bernstein and N. Goodman, An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases, *Transactions on Database Systems 9*, 4 (December 1984), 596-615.

[Bla83] A. P. Black, An Asymmetric Stream Communication System, *Proceedings 9th Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, 10-13 November 1983, 4-10.

[Bri69] P. Brinch-Hansen, ed., *RC4000 Software: Multiprogramming System*, Regnecentralen, Copenhagen, Denmark, April 1969.

[Bri73] P. Brinch-Hansen, *Operating Systems Principles*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[Che82] D. R. Cheriton, *The Thoth System: Multiprocess Structuring and Portability*, Elsevier-North Holland, New York, 1982.

[Che84] D. R. Cheriton, The V Kernel: A Software Base for Distributed Systems, *IEEE Software 1*, 2 (April 1984), 19-42.

[Coo84] E. C. Cooper, Circus: A Replicated Procedure Call Facility, *Proceedings 4th Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, 15-17 October 1984, 11-23.

[CrE84] D. Crookes and J. W. G. Elder, An Experiment in Language Design for Distributed Systems, *Software Practice and Experience 14*, 10 (October 1984), 957-971.

[Dec87] K. S. Decker, Distributed Problem-Solving Techniques: A Survey, *IEEE Transactions on Systems, Man, and Cybernetics SMC-17*, 5 (September/October 1987), 729-740.

[Dij68] E. W. Dijkstra, The Structure of THE Multiprogramming System, *Communications of the ACM 11*, 5 (May 1968), 341-346.

[Dou84] G. Doumeingts, Methodology to Design Computer Integrated Manufacturing and Control of Manufacturing Units, in *Methods and Tools for Computer Integrated Manufacturing*, U. Rembold and R. Dillman (ed.), Springer-Verlag, New York, 1984. Advanced CREST Course on CIM (CIM 83), Karlsruhe, Federal Republic of Germany, 5-16 September 1983.

[EFH82] C. S. Ellis, J. A. Feldman and J. E. Heliotis, Language Constructs and Support Systems for Distributed Computing, Tech. Rep. 102, Department of Computer Science, University of Rochester, May 1982.

[Eri82] L. W. Ericson, DPL-82: A Language for Distributed Processing, *Proceedings 3rd International Conference on Distributed Computing Systems*, Ft. Lauderdale, Florida, 18-22 October 1982, 526-531.

[EFR86] G. Estrin, R. S. Fenchel, R. R. Razouk and M. K. Vernon, SARA (System ARchitects Apprentice): Modelling, Analysis and Simulation Support for Design of Concurrent Systems, *IEEE Transactions on Software Engineering SE-12*, 2 (February 1986), 293-311.

[Fel79] J. A. Feldman, High Level Programming for Distributed Computing, *Communications of the ACM 22*, 6 (June 1979), 353-368.

[FHH87] R. L. Franks, J. P. Holtman, J. L. C. Hsu, L. G. Raymer and B. E. Snyder, Productivity Improvement Systems for Manufacturing, *AT&T Technical Journal 66*, 5 (September/October 1987), 61-76.

[Fri86] S. A. Friedberg, User Process - HPC Interface, HPC Project Report 3, University of Rochester, October 1986.

[FrP86] S. A. Friedberg and D. H. Pitcher, HPC IPC Implementation, HPC Project Report 2, University of Rochester, June 1986.

[Fri87] S. A. Friedberg, IPC for Modular Software Requires a Third Party Connect, Tech. Rep. 220, University of Rochester, June 1987.

[Hel84] J. Heliotis, Language Constructs for the Management of Distributed Computations, Ph.D. Thesis, University of Rochester, 1984.

[Hoa85] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[HoR73] J. J. Horning and B. Randell, Process Structuring, *Computing Surveys 5*, 1 (March 1973), 5-30.

[IEE75] IEEE, Digital Interface for Programmable Instrumentation, April 1975.

[JBH78] I. M. Jacobs, R. Binder and E. V. Hoversten, General Purpose Packet Satellite Networks, *Proceedings of the IEEE 66*, 11 (November 1978), 1448-1467.

[JCD79] A. K. Jones, R. J. Chansler, I. Durham, K. Schwan and S. R. Vegdahl, StarOS, A Multiprocessor Operating System for the Support of Task Forces, *Proceedings 7th Symposium on Operating Systems Principles*, Pacific Grove, California, Dec 1979, 117-127.

[Jon79] A. K. Jones, Protection Mechanisms and the Enforcement of Security Policies, in *Operating Systems, An Advanced Course*, R. Bayer, R. M. Graham and G. Seegmueller (ed.), Springer-Verlag, New York, 1979, 228-251.

[JuT87] J. Jubin and J. D. Tornow, The DARPA Packet Radio Network Protocols, *Proceedings of the IEEE 75*, 1 (January 1987), 21-32.

[KGB78] R. E. Kahn, S. A. Gronemeyer, J. Burchfiel and R. C. Kunzelman, Advances in Packet Radio Technology, *Proceedings of the IEEE 66*, 11 (November 1978), 1468-1496.

[KrM85] J. Kramer and J. Magee, Dynamic Configuration for Distributed Systems, *IEEE Transactions on Software Engineering SE-11*, 4 (April 1985), 424-436.

[KMS87] J. Kramer, J. Magee and M. Sloman, The CONIC Toolkit for Building Distributed Systems, *IEE Proceedings 134-D*, 2 (March 1987), 73-82.

[LMW86] R. P. Laesar, W. I. McLaughlin and D. M. Wolff, Engineering Voyager 2's Encounter with Uranus, *Scientific American*, September 1986, 36.

[Lam78] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM 21*, 7 (July 1978), 558-564.

[LeM82] R. J. LeBlanc and A. B. Maccabe, The Design of a Programming Language Based on Connectivity Networks, *Proceedings 3rd International Conference on Distributed Computing Systems*, Ft. Lauderdale, Florida, 18-22 October 1982, 532-541.

[LeF85] T. J. LeBlanc and S. A. Friedberg, HPC: A Model of Structure and Change in Distributed Systems, *IEEE Transactions on Computers C-34*, 12 (December 1985), 1114-1129.

[LeW85] R. J. LeBlanc and C. T. Wilkes, Systems Programming with Objects and Actions, *Proceedings 5th International Conference on Distributed Computing Systems*, Denver, Colorado, 13-17 May 1985, 132-139.

[LeF85] T. J. LeBlanc and S. A. Friedberg, Hierarchical Process Composition in Distributed Operating Systems, *Proceedings 5th International Conference on Distributed Computing Systems*, Denver, Colorado, 13-17 May 1985, 26-34.

[LiS83] B. Liskov and R. Scheifler, Guardians and Actions: Linguistic Support for Robust, Distributed Programs, *Transactions on Programming Languages and Systems 5*, 3 (July 1983), 381-404.

[Loc79] T. W. Lockhart, The Design of a Verifiable Operating System Kernel, CS-Tech. Rep. 79-15, University of British Columbia, November 1979.

[LoA78] D. C. Loughry and M. S. Allen, IEEE Standard 488 and Microprocessor Synergism, *Proceedings of the IEEE 66*, 2 (February 1978), 162-172.

[Mar84] K. Marzullo, Loosely-Coupled Distributed Services: A Distributed Time Service, Ph.D. Thesis, 1984.

[MFR78] J. M. McQuillan, G. Falk and I. Richer, A Review of the Development and Performance of the ARPANET Routing Algorithm, *IEEE Transactions on Communication COM-26*, 12 (December 1978), 1802-1810.

[MRR80] J. M. McQuillan, I. Richer and E. C. Rosen, The New Routing Algorithm for the ARPANET, *IEEE Transactions on Communication COM-28*, 5 (May 1980), 711-719.

[Moc87a] P. Mockapetris, Domain Names - Concepts and Facilites, Request for Comments 1034, DARPA Network Working Group, November 1987.

[Moc87b] P. Mockapetris, Domain Names - Implementation and Specification, Request for Comments 1035, DARPA Network Working Group, November 1987.

[MuJ78] T. Mukaihata and R. D. Johnstone, Implementation and Use of Small Automated-Test Systems, *Proceedings of the IEEE 66*, 4 (April 1978), 403-413.

[Ous81] J. K. Ousterhout, *Medusa: A Distributed Operating System*, UMI Research Press, Ann Arbor, Michigan, USA, 1981.

[PeB79] M. H. Penedo and D. M. Berry, The Use of a Module Interconnection Language in the SARA System Design Methodology, *Proceedings 4th International Conference on Software Engineering*, Munich, Federal Republic of Germany, 17-19 September 1979, 294-307.

[PWC81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin and G. Thiel, LOCUS: A Network Transparent, High Reliability Distributed System, *Proceedings 8th Symposium on Operating Systems Principles*, Pacific Grove, California, 14-16 December 1981, 169-177.

[Ran79] B. Randell, Reliable Computing Systems, in *Operating Systems, An Advanced Course*, R. Bayer, R. M. Graham and G. Seegmueller (ed.), Springer-Verlag, New York, 1979, 282-391.

[Ran83] P. Ranky, *The Design and Operation of FMS*, North-Holland, New York, 1983. (Flexible Manufacturing Systems).

[Ray87] M. Raynal, A Distributed Algorithm to Prevent Mutual Drift Between N Logical Clocks, *Information Processing Letters 24*, 3 (13 February 1987), 199-202.

[Rid81] W. E. Riddle, An Assessment of DREAM, in *Software Engineering Environments*, H. Hunke (ed.), North-Holland, 1981, 191-221.

[Ros85] D. T. Ross, Applications and Extensions of SADT, *Computer 18*, 4 (April 1985), 25-34.

[SBN83] M. D. Schroeder, A. D. Birrell and R. M. Needham, Experience with Grapevine: The Growth of a Distributed System, *Proceedings 9th Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, November 1983, 16-37.

[SBW87] K. Schwan, T. Bihari, B. W. Weide and G. Taulbee, High-Performance Operating System Primitives for Robotics and Real-Time Control Systems, *Transactions on Computer Systems 5*, 3 (August 1987). 189-231.

[ScY88] M. L. Scott and S. Yap, A Grammar-Based Approach to the Automatic Generation of User-Interface Dialogues, *Proceedings Computer-Human Interface '88 Conference*, May 1988, 73-78.

[ShW88] M. J. Shaw and A. B. Whinston, A Distributed Knowledge-Based Approach to Flexible Automation: The Contract Net Framework, *International Journal of Flexible Manufacturing Systems 1*(1988), 85-104.

[Sim62] H. A. Simon, The Architecture of Complexity, *Proceedings American Philosophical Society 106*(1962), 476-482. reprinted in *The Sciences of the Artificial*, MIT Press, Cambridge, Massachussetts, 1969.

[Smi78] R. G. Smith, Applications of the Contract Net Framework: Distributed Sensing, *Proceedings DARPA Distributed Sensor Net Symposium*, Pittsburgh, Pennsylvania, December 1978, 12-20.

[Smi80] R. G. Smith, Applications of the Contract Net Framework: Search, *Proceedings 3rd Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, Victoria, British Columbia, 14-16 May 1980, 232-239.

[StS75] J. Staunstrup and S. M. Sorenson, Platon - A High-Level Language for Systems Programming, R-75-59, RECAU Aarhus University, May 1975.

[SLR76] R. E. Stearns, P. M Lewis and D. J. Rosenkrantz, Concurrency Control for Database Systems, *Proceedings 16th IEEE Conference on Foundations of Computer Science*, October 1976, 19-32. CH1133-8C.

[Sto77] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Massachusetts, 1977.

[SFS77] R. J. Swan, S. H. Fuller and D. P. Siewiorek, Cm* - A Modular Multi-Microprocessor, *Proceedings 1977 AFIPS National Computer Conference 46*(?? 1977), 637-644.

[SBL77] R. J. Swan, A. Bechtolsheim, K. Lai and J. K. Ousterhout, The Implementation of the Cm* Multi-Microprocessor, *Proceedings 1977 AFIPS National Computer Conference 46*(?? 1977), 645-655.

[TaW82] R. Taylor and P. Wilson, Process-Oriented Language Meets Demands of Distributed Computing, *Electronics 55*, 24 (30 November 1982), 89-95.

[WPE83] B. J. Walker, G. J. Popek, R. English, C. Kline and G. Thiel, The LOCUS Distributed Operating System, *Proceedings 9th Symposium or Operating Systems Principles*, Bretton Woods, New Hampshire, 10-13 October 1983, 49-70.

[WeL88] J. L. Welch and N. Lynch, A New Fault-Tolerant Algorithm for Clock Synchronization, *Information and Computation 77*, 1 (April 1988), 1-36.

[YTR87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black and R. Baron, The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, *Proc. 11th Symp. on Operating System Prin.*, Austin, Texas, Nov 1987, 63-76.

[Yue83] J. H. Yuen, ed., *Deep Space Telecommunications Systems*, Plenum Press, New York, 1983.

[ZwA85] W. Zwaenepoel and G. T. Almes, Understanding and Exploiting Distribution, Technical Report 85-12, Department of Computer Science, Rice University, February 1985.